TÉCNICO LISBOA

Plataforma Fog Distribuída para Previsões Meteorológicas em Edge

Alexandre Bruno Gehl Baptista Rodrigues da Costa

Dissertação para obtenção do Grau de Mestre em

Engenharia de Telecomunicações e Informática

Orientador: Prof. Luís Manuel Antunes Veiga

Júri

Presidente: Prof. Ricardo Jorge Fernandes Chaves
Orientador: Prof. Luís Manuel Antunes Veiga
Vogal: Prof. Rolando da Silva Martins

Novembro 2025

Declaração
Declaro que o presente documento é um trabalho original da minha autoria e que cumpre todos os requisitos do Código de Conduta e Boas Práticas da Universidade de Lisboa.

Agradecimentos

A realização deste trabalho não teria sido possível sem o apoio e contributo de várias pessoas, às quais gostaria de expressar o meu mais sincero agradecimento.

Em primeiro lugar, agradeço profundamente à minha família, mãe, pai, irmã e avós, e também à minha saudosa comparsa de quatro patas, que me acompanhou em todo este caminho, pelo apoio constante, pela paciência nos momentos mais exigentes e pela motivação que sempre me transmitiram ao longo de todo o percurso académico.

Aos meus professores e colegas do curso, deixo o meu reconhecimento pelo conhecimento partilhado, pela disponibilidade e pelas discussões construtivas que tanto contribuíram para o meu crescimento académico e pessoal.

Um agradecimento especial ao Professor Luís Veiga, pela orientação atenta, pelo incentivo contínuo e pelas sugestões valiosas durante todas as fases do desenvolvimento desta dissertação. A sua experiência e acompanhamento foram determinantes para a concretização deste trabalho.

A todos os que, de forma direta ou indireta, contribuíram para esta etapa da minha vida, deixo o meu mais sincero obrigado.

Resumo

Com o crescente desenvolvimento e adoção de tecnologias Internet of Things (IoT), torna-se necessário criar condições para que estas possam funcionar de forma eficaz e escalável. A quantidade de dados gerada e transferida constantemente por este tipo de dispositivos é extremamente elevada, sendo que muitos destes sistemas podem suportar serviços que operam em tempo real, e com elevado sentido de urgência (por exemplo, tecnologias relacionadas com contextos hospitalares/saúde ou com redes veiculares inteligentes). Torna-se por isso essencial encontrar soluções e vias que possibilitem e otimizem o seu funcionamento, e que consigam lidar com estes grandes fluxos de dados, seja a nível de armazenamento, processamento ou transferência. A computação em edge, um paradigma que desloca o processamento desses dados para junto do utilizador, resolve alguns destes problemas, principalmente reduzindo latências de resposta, e oferecendo também vantagens em termos de privacidade. Se o dispositivo IoT for capaz de realizar todo ou parte do processamento necessário localmente, evita, desde logo, que a totalidade dos dados tomem caminhos para fora da rede local, já que podem ser tratados próximos de onde foram gerados/obtidos. Apesar de tudo, é legítimo assumir que, mesmo que estes dispositivos possam realizar as tarefas necessárias independentemente, seja necessário um mecanismo superior que seja capaz de distribuir essas mesmas tarefas e organizar o esforço computacional exigido. Neste contexto, surge como solução o paradigma da computação em fog, que propõe uma camada intermédia, localizada acima dos nós locais em edge, capaz de organizar o processamento distribuído de forma coordenada. Esta arquitetura, ao aproximar a inteligência computacional do utilizador, permite não só segmentar e distribuir as tarefas de forma eficaz, como também reforça a relevância da computação distribuída em cenários com requisitos críticos de desempenho. Este trabalho discute a atualidade e a pertinência destes paradigmas, propondo e desenvolvendo uma plataforma que visa demonstrar o seu potencial. Esta plataforma, foca-se na previsão de variáveis meteorológicas — como temperatura, humidade relativa, velocidade do vento ou pressão atmosférica — utilizando tecnologias orientadas a processamento distribuído e big data como Apache Spark e Hadoop para suportar

o cluster.

Palavras Chave

Computação em Edge; Arquitetura Fog; Internet of Things; Computação Distribuída

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

Acrónimos

ACF	Autocorrelation Function
AIC	Akaike Information Criterion
ANN	Artificial Neural Network
ARIMA	AutoRegressive Integrated Moving Average
CNN	Convolutional Neural Network
HDFS	Hadoop Distributed File System
IoT	Internet of Things
MAPE	Mean Absolute Percentage Error
MAE	Mean Absolute Error
MSE	Mean-Square Error
NWP	Numerical Weather Predictions
PACF	Partial Autocorrelation Function
RMSE	Root Mean-Square Error
RNN	Recurrent Neural Network
SARIMA	Seasonal AutoRegressive Integrated Moving Average

1

Introdução

Conteúdo

A computação em *cloud* tem-se consolidado como um dos paradigmas mais populares e essenciais da evolução tecnológica recente, principalmente devido à escalabilidade e aos variados conjuntos de serviços disponíveis (sejam arquiteturas SaaS - *Software as a Service* - ou PaaS, *Platform as a Service*; ou ainda IaaS, ou *Infrastructure as a Service*) capazes de responder às mais diversas necessidades, seja a nível de recursos físicos ou virtuais que oferecem, e permitindo que as organizações/utilizadores se alavanquem desses recursos sem terem a necessidade de mantê-los em infraestruturas próprias.

A computação em *edge*, próxima do utilizador, e a computação em *fog*, que providencia recursos importantes na fronteira entre a rede local edge e a cloud [1], vêm contribuir significativamente para solucionar os desafios descritos em capítulo seguinte, e levantados por estas arquiteturas de computação. Este paradigma *edge* implica um modelo em que o processamento de dados ocorre junto ao utilizador ou à fonte de dados, focando-se na descentralização dos recursos computacionais, e permitindo que dispositivos que tradicionalmente não serviriam para tal intuito, passem agora a poder contribuir com-

putacionalmente *in loco* para o sistema. Já a componente *fog*, posicionada entre a edge e a cloud, facilita a transferência e a otimização de recursos para o processamento dos dados, distribuindo cargas de trabalho de maneira inteligente, considerando as condições de rede e as condições e capacidades dos dispositivos ao seu dispôr. Estes paradigmas, ao resolverem problemas críticos de latência, escalabilidade e *bandwidth* [1], não só aumentam a eficiência operacional de um sistema neste ambiente, como ainda possibilitam o surgimento de novas aplicações e serviços que anteriormente se debatiam com os desafios referidos.

1.1 Desafios

Apesar do sucesso e importância destas tecnologias e deste modelo de computação, a abordagem *cloud* não parece estabelecer o melhor paradigma para lidar com quantidades massivas de dados transferidos em tempo real, pois a sua transferência implica atrasos consideráveis, potencialmente escalando em função do contexto. Principalmente desde o surgimento das tecnologias Internet of Things (que geram as tais quantidades enormes de dados em tempo real e requerem muitas vezes decisões também em tempo real) que se tem explorado vias para aproximar o processamento e armazenamento do utilizador [2]. A computação em cloud, devido a essas limitações, tem dificuldades e é incapaz de lidar por si só com as crescentes exigências a que as tecnologias IoT obrigam [3], principalmente ao nível de serviços críticos como aqueles relacionados com sistemas de saúde, redes veiculares, monitoramento ambiental ou *smart cities*. Além disso, as crescentes necessidades de capacidade de processamento em tempo real e de minimização de latência, tornam estas arquiteturas pouco eficientes a lidar com estes cenários que exigem respostas e decisões quase instantâneas. Assim, as soluções atuais baseadas exclusivamente em infraestruturas cloud apresentam limitações significativas quando aplicadas a cenários que exigem processamento em tempo real ou elevada escalabilidade. Por outro lado, a utilização de dispositivos com recursos computacionais reduzidos — como dispositivos móveis ou sensores — levanta desafios adicionais relacionados com a capacidade de processamento e estabilidade da rede.

1.2 Objetivos

Sendo que estas enormes e complexas redes que servem de esqueleto a uma cidade inteligente, constituídas por diferentes tecnologias Internet of Things (IoT) que permitem comunicações e transferências de dados entre qualquer tipo de objetos ou dispositivos, produzem quantidades de dados massivas [4], torna-se necessário aliviar a carga que passa para a rede core, até para dar cumprimento às rápidas respostas que muitos destes dispositivos e serviços necessitam.

Este trabalho propõe-se inicialmente a realizar um processo de análise do 'estado da arte' das temáticas e conceitos já introduzidos e outros relacionados, estudando e comparando algumas abordagens de projetos ou investigações nestas áreas (como *smart cities* ou a *Internet of Things* e o processamento de *big data*), identificando problemas comuns e possíveis soluções. Com base nestas análises, é esboçada uma aplicação e uma arquitetura que se utilize destas componentes e de *features* de paradigmas e modelos apontados para as mesmas (como computação em *fog* e *edge*, ou processamento distribuído), contornando as dificuldades comumente encontradas em aplicações desenvolvidas nestes contextos de cidades inteligentes ou que tratem de volumes elevados de dados. O intuito desta plataforma, a qual iremos denominar por Weather@Edge, será pôr à disposição e próximos do utilizador, serviços que habitualmente só estariam disponíveis através da cloud. Assim, esta arquitetura fornece independência ao utilizador, reduz o tráfego de dados, aproveita recursos computacionais que anteriormente não seriam utilizados, e dilui esforços computacionais pelos nós disponíveis e capazes de contribuir. Portanto, tira partido duma rede de dispositivos locais, e descongestiona as redes que hoje em dia têm de suportar este tipo de serviços. Uma arquitetura de computação em edge será então a base deste tipo de plataformas, passando a componente de computação dos serviços e idealmente todo o serviço que habitualmente estaria na cloud, para próximo dos utilizadores mobile. Ao contrário de uma arquitetura full-edge a 100%, que implicaria também que os dados relevantes fossem recolhidos por sensores locais, e armazenados, por exemplo, nos próprios sensores (ou em infra-estruturas/dispositivos próximos do utilizador), no caso da arquitetura da nossa plataforma, pretendemos sim distribuir a carga de processamento, mas também garantir que os dados de input para o processamento são obtidos a partir de repositórios validados (ou que sejam mantidos pela própria aplicação, ou que estejam acessíveis a partir de URLs externos de instituições que disponibilizem dados climáticos históricos). De seguida, é iniciado o processo de desenvolvimento de um sistema assente na plataforma e estratégia discutida e que mostre melhorias de eficiência a uma arquitetura centralizada sem capacidades de execução paralela.

Assim, este documento, após definir os objetivos e limitações associados ao tema, irá começar por analisar trabalhos relacionados à área de estudo. A partir dessa base, sugerirá uma solução para um contexto concreto aplicável a esta mesma área em análise, descrevendo a implementação e configuração dessa solução. Segue-se a avaliação da mesma, aferindo se esta responde positivamente aos objetivos que identificámos. Por fim, irão tirar-se conclusões sobre toda a investigação e aplicação da proposta.

Quanto à avaliação da solução, pretende-se apresentar metodologias de avaliação aplicáveis à plataforma, seja ao nível da avaliação dos modelos de previsão utilizados, ou ao nível da avaliação da eficácia da orquestração ou distribuição e da cooperação dos elementos pertencentes a uma plataforma de computação em paralelo e distribuída. Seguindo as diretrizes definidas, passar-se-á então

para a avaliação do sistema concebido, tirando daí conclusões que possam contribuir para o futuro deste tipo de soluções.

1.3 Contribuições

Tirando partido destas conclusões, foi desenvolvida uma plataforma, que conjuga as abordagens *edge* e *fog*, e é capaz de realizar previsões meteorológicas, fazendo uso, não só dos recursos dos dispositivos do utilizador, como também de dispositivos das redondezas que se apresentem como capazes de contribuir. Estes dispositivos são orquestrados por uma camada superior de *fog*, que irá organizar e distribuir o esforço cooperativo. Esta combinação de capacidade de processamento local, com a inteligência e flexibilidade que a camada *fog* traz, proporciona um equilíbrio ideal para lidar com estas exigências da IoT e das *smart cities*. Iremos portanto explorar e testar como estas arquiteturas funcionam em conjunto num contexto prático em ambiente controlado. No caso, irá calcular-se a previsão de variáveis meteorológicas (temperatura, humidade e precipitação) horárias para um intervalo próximo de tempo. Abordaremos também questões relacionadas com distribuição de dados, otimização de recursos, e integração entre os diferentes níveis de computação, oferecendo uma perspetiva de como a computação *edge* e *fog*, conjugadas, podem melhorar significativamente a eficiência do processamento de dados, diminuir os tempos de latência e reduzir a sobrecarga da rede na nuvem (*cloud)*, permitindo que um determinado sistema tome decisões rápidas e precisas.

O trabalho desenvolvido iniciou-se com o estudo e análise do estado da arte, onde foram investigados os principais paradigmas de computação — cloud, fog e edge —, bem como as abordagens recentes aplicadas à previsão meteorológica e à gestão de dados em ambientes urbanos inteligentes. A partir dessa análise, foi concebido o desenho de uma arquitetura distribuída baseada em fog e edge computing, projetada para suportar o processamento descentralizado de dados meteorológicos e permitir a execução de algoritmos de previsão de forma eficiente e escalável.

De seguida, procedeu-se à implementação da aplicação Weather@Edge, integrando módulos de parametrização e previsão que utilizam mecanismos de processamento paralelo e colaborativo. Por fim, foi realizada a avaliação do desempenho e da eficiência da aplicação desenvolvida, recorrendo a métricas relacionadas com latência, tempo de execução, consumo de recursos e escalabilidade.

Desta forma, este trabalho contribui para a compreensão e validação do potencial das arquiteturas fog e edge na execução de workloads de previsão meteorológica, demonstrando que a descentralização do processamento pode reduzir significativamente a sobrecarga de rede e os tempos de resposta, permitindo que sistemas em smart cities tomem decisões de forma mais rápida e precisa.

2

Trabalhos Relacionados

Conteúdo

Com os avanços das tecnologias IoT ou *machine learning*, surge a necessidade de explorar diferentes abordagens que permitam perceber as maneiras mais eficientes de lidar com as mesmas a nível computacional e de conjugá-las. Inicialmente, será explorado o conceito de *smart cities*, analisando a forma como a computação em edge pode ser aplicada para dar resposta à elevada demanda de dados em tempo real em cenários deste tipo. De seguida, a arquitetura fog será aprofundada, percebendo a sua importância como camada 'orquestradora' da camada de nós locais, ou até como elo de ligação entre essa rede local e a cloud. O tópico sobre computação distribuída pretende perceber as tecnologias popularmente utilizadas para projetos similares, percebendo e relevando a sua importância em plataformas que lidem com processamentos que sejam exigentes ou que lidem com grandes quantidades de dados. Por fim, são estudados alguns sistemas de previsões meteorológicas e métodos tradicionais para desenvolver aplicações desse tipo.

Esta revisão visa então identificar lacunas nas soluções propostas e existentes, tal como os diferentes elementos positivos que as constituem e que as tornam interessantes. Assim, estabelecemos uma base para o desenvolvimento da proposta apresentada no capítulo 3.

2.1 Smart Cities @ Edge

O já referido contínuo desenvolvimento de tecnologias IoT, acaba por fortalecer e despertar curiosidade para o conceito de smart cities, ou cidades inteligentes. A potencialidade presente em tecnologias tradicionais é imensa, e os exemplos são variadíssimos:

- Urbanizações repletas de redes de infraestruturas inteligentes, que comunicam entre si e recolhem e partilham informações, dados, padrões, tudo aquilo que possa oferecer valências às cidades e aos seus habitantes [5].
- Semáforos capazes de reconhecer padrões de tráfego, ou identificar veículos via streams de vídeo, e capazes de tomar decisões autonomamente, e de eles próprios armazenarem dados obtidos ou decisões tomadas [6].
- Iluminação pública interconectada e capaz de tomar decisões baseando-se nos dados partilhados entre si ou em dados recolhidos por eles via sensores [7].
- Automóveis que calculam rotas com base no tráfego daquele dia, ou redes veiculares capazes de funcionar autonomamente [8, 9].
- Dados locais das mais variadas naturezas (por exemplo temperatura, humidade, ou probabilidade de precipitação num determinado local), disponíveis localmente para consulta pública por parte dos cidadãos/utilizadores, por exemplo através de pedido API, sem necessidade de recurso a cloud ou servidores centrais (já explorando um pouco o conceito de tecnologias da informação e comunicação) [10].
- Câmaras de vigilância espalhadas pelas cidades inteligentes a gerar continuamente dados e capazes de identificar autonomamente situações de perigo, ou aplicações smart healthcare, como sensores que monitorizam continuamente determinada condição de saúde de um paciente, enviando os dados para a cloud para estes serem processados [11, 12].

Estes cenários são apenas exemplos de como tecnologias habituais do nosso dia-a-dia podem “ganhar inteligência”, e como uma cidade se pode transformar numa smart city, e oferecer novas e variadas valências, ou melhorar serviços e tecnologias já existentes, maximizando as funcionalidades das estruturas que nos rodeiam. Os aumentos de eficiência em serviços complexos que funcionam em tempo

real, poupanças energéticas [13], as reduções de congestionamentos de redes, e os benefícios para os cidadãos poderiam ser enormes. Os objetivos das smart cities estão alinhados com a melhoria da qualidade de vida dos cidadãos e a eficiência urbana. Por isso, os principais vetores explorados neste tipo de ecossistemas, acabam por ser a mobilidade urbana, a redução dos consumos energéticos (através de, por exemplo, tecnologias smart grid, plataformas inteligentes capazes de monitorar e otimizar autonomamente os fluxos energéticos nas redes elétricas [14]), a melhoria do acesso e da qualidade de serviços públicos, e a promoção da sustentabilidade ambiental (seja através de sistemas de controlo e alerta para desastres naturais, sensores que monitorizam permanentemente a qualidade do ar, ... [15]).

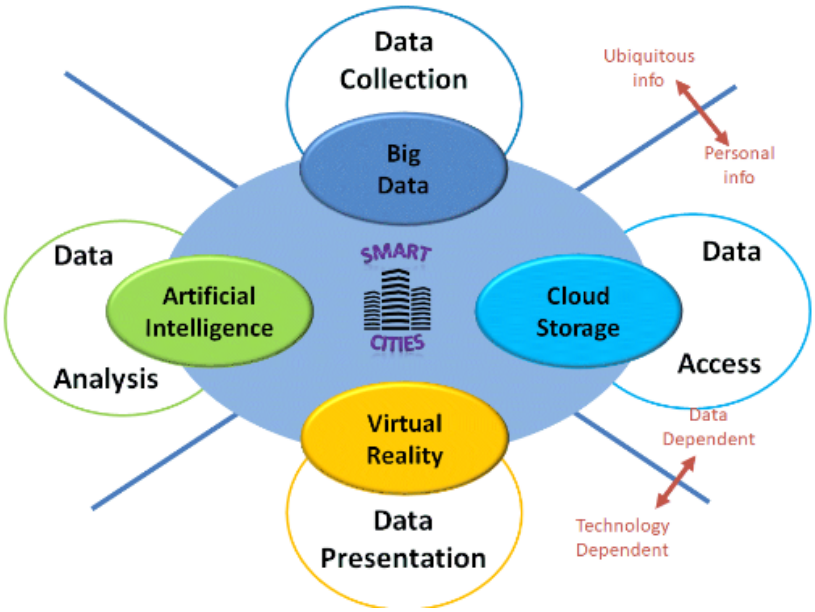


Figura 2.1: Framework para smart cities da Global Development research Center

Nam e Pardo [16] destacam que tecnologias relacionadas com IoT, big data e inteligência artificial, são um passo importante e que permitem a conexão entre os diferentes serviços urbanos, e otimizam a distribuição de recursos. Esta "inteligência" nas tecnologias, pressupõe o princípio da computação automática, o que significa que essa mesma tecnologia terá de ser capaz de, autonomamente, realizar tarefas de configuração (self-configuration), recuperação de erros (self-healing), e otimização (self-optimization).

Obviamente que este tipo de benefícios traz também consigo novas exigências. A quantidade de dados gerados por estas tecnologias é enorme. Muitos deles com relevância suficiente para justificarem e requerem o seu armazenamento. Para além disso, são tecnologias e dispositivos que estão a comunicar e a oferecer serviços que exigem respostas e decisões em tempo crítico e com urgência. Como referem Apat et al. [17], o modelo de computação em cloud, apesar de oferecer uma vasta pool

de recursos físicos ou virtuais, não apresenta os resultados ideais a nível de latência para utilizadores IoT, devido ao extenso fluxo de tráfego dos dados, e às possíveis limitações em termos de largura de banda da rede provocados pela excessiva concorrência de pedidos efetuados num dado momento [17]. Como lidar então com estes desafios?

Deslocando tanto o processamento, como o armazenamento dos dados, para quão próximo seja possível dos próprios dispositivos que estão a gerar os dados. É aqui que surge a componente "@ Edge", como é referido no título da seccção, ou "na periferia". Estudos revelam que a computação em edge é a solução ideal quando os objetivos são reduzir latências e melhorar a privacidade dos dados [18], devido à redução do caminho que estes têm de percorrer ao serem processados localmente. Olhando para o cenário idílico nesse tipo de arquitetura, não haveria necessidade de haver comunicação com a cloud em nenhum momento. Não sendo necessário existir ligação entre os nós locais e a cloud, os nós teriam de obter os dados necessários desde tecnologias (sensores ou outro tipo de estruturas capazes de armazenar os dados localmente) de recolha locais.

2.2 Arquitetura Fog

Apesar dos benefícios que já discutimos e que a computação em edge pode oferecer, há que considerar também que é uma realidade que este tipo de arquiteturas pode dificultar a escalabilidade e flexibilidade de um sistema; exatamente o oposto daquilo que as arquiteturas baseadas em cloud oferecem, portanto.

Surge então uma solução híbrida capaz de providenciar essa 'elasticidade' à borda da rede e de resolver estas dificuldades identificadas [1] , e que faz uso de componentes de serviços edge, mas também de serviços cloud: a arquitetura *fog*, ou em nevoeiro. A camada fog, que pode ser definida como a camada que faz a intermediação entre os nós locais e a cloud, à distância de 1 hop dos nós, acaba por ser combinar valências de ambas as abordagens, oferecendo baixas latências e sendo eficaz na gestão de recursos e largura de banda disponível [18]. Os dispositivos em edge, geram e processam os dados em tempo real, ficando a responsabilidade de orquestração e coordenação dos nós, ao encargo desta camada. Tipicamente, este tipo de arquitetura oferece também capacidade de offloading, distribuindo e atribuindo tarefas com base, por exemplo, na capacidade computacional, na bateria, ou na proximidade entre dispositivos.

Bebortta et al. [19], desenharam uma framework para otimização de tratamento de big data em redes IoT heterogéneas que "permite recolha de dados, feature selection, modelos preditivos e visualização de dados", e assenta essencialmente numa estrutura do tipo que é possível observar na Figura 2.2.

No diagrama ilustrado na Figura 2.2, representativo da arquitetura geral, pode-se verificar que existe também uma camada de gateways entre os nós locais e a camada fog, devido à heterogeneidade de dis-

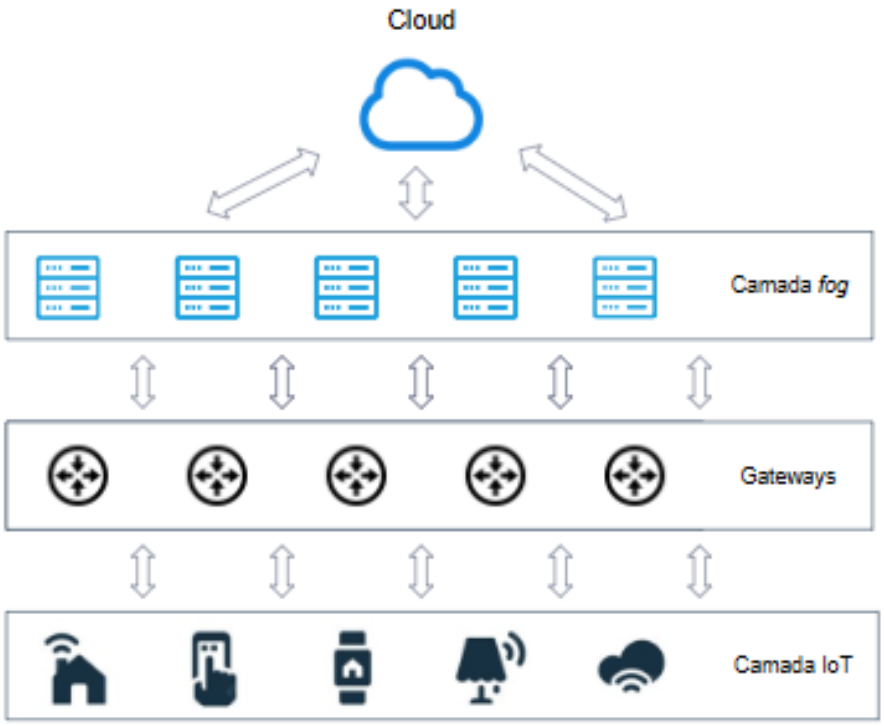


Figura 2.2: Arquitetura Fog

positivos que podem interagir com o serviço, tornando-se necessário a existência de homogeneização dos protocolos/comunicações. Esta camada pode ser fundamental tanto na orquestração da distribuição dos esforços computacionais como, em caso de necessidade, auxiliando nesses próprios esforços. Cada nó terá sempre um dispositivo fog a si associado, ao qual nos iremos referir como local server ou LS (cada um responsável pelo seu cluster de nós locais), e cujo papel poderá ser desempenhado por nós também eles próximos do utilizador, como routers, switches, access points ou servidores locais. Num dado momento, cada nó local poderá ter a si associado apenas um local server, sendo que estes poderão ter associados a si múltiplos nós locais, ficando responsáveis pela sua orquestração. Este paradigma acaba por relevar a componente de descentralização desta arquitetura, oferecendo redundância aos sistemas e aumentando a sua capacidade de tolerância a erros. Em caso de falha de um dos nós, a camada fog, ou o local server em questão, terá de ter isso em consideração e possivelmente redistribuir as cargas anteriormente atribuídas aos seus nós locais. É por isso expectável que estes nós tenham de anunciar periodicamente a sua disponibilidade ao local server associado.

Ao nível de segurança e privacidade, as vantagens podem também ser inúmeras: os dados que podem agora ser processados localmente, não têm de ser transferidos para além da rede local, reduzindo os riscos associados ao tráfego de dados na rede e às possíveis vulnerabilidades da cloud. Em termos de escalabilidade, esta arquitetura distribuída facilita a adição de nós a um sistema em caso de aumento

da necessidade de capacidade de processamento ou armazenamento, tornando-se assim um modelo muito apetecível para redes de sensores de larga escala, como no contexto das cidades inteligentes, onde o número de dispositivos locais e a produção de dados pode aumentar constantemente.

2.3 Computação Distribuída

Considerando a já mencionada crescente necessidade de processar enormes quantidades de dados em tempo real, para além da importância da proximidade do processamento e do armazenamento dos dados - a componente edge portanto - é importante também destacar os ganhos e vantagens na implementação de um tipo de arquitetura que possibilite aos nós orquestrar e distribuir entre si aquilo que é exigido ao sistema a nível de esforço. A parte da orquestração poderá ser executada, como já referimos, pelos elementos que constituem a camada "superior" aos nós, a camada fog. A computação distribuída permite assim aos sistemas, dividirem uma determinada tarefa em segmentos mais pequenos, que podem depois ser distribuídos por diferentes dispositivos capazes de executá-los. Torna-se, portanto, uma solução bastante viável para contextos que exijam constante ou grande esforço computacional, como aqueles presentes nas cidades inteligentes que temos vindo a falar [20].

Tendo então conferido à arquitetura capacidade de orquestração dos nós constituintes da rede, através dessa camada intermédia, então, caso as tarefas necessárias ao funcionamento do sistema, forem ainda particionadas e distribuídas por diferentes nós capazes de contribuir, para além da proximidade que conquistámos entre esse mesmo sistema e o utilizador, conseguimos também descentralizá-lo totalmente e, assim, optimizar a sua eficiência.

Havendo sempre um mínimo de 2 elementos no sistema quando uma execução é realizada, o nó utilizador e o respetivo LS 'orquestrador', da camada fog, então, isso significa que existirá sempre paralelismo e segmentação dos cálculos. Para suportar essa divisão de cálculos, podem ser exploradas *frameworks* como o Apache Hadoop e o Spark. O Hadoop, uma biblioteca *open-source* amplamente utilizada neste tipo de contextos de computação de grandes volumes de dados [21, 22] (também estendido para computação voluntária [23, 24]), através do seu sistema de arquivos distribuído, o HDFS (*Hadoop Distributed File System*), permite o armazenamento distribuído de conjuntos de dados entre os nós da rede [21]. Outro pilar importante na framework do Hadoop, é o ficheiro YARN (*Yet Another Resource Negotiator*), que funciona como o 'gestor' dos recursos e da orquestração de tarefas em clusters distribuídos, oferecendo flexibilidade ao Hadoop que inicialmente assentava apenas em modelos MapReduce. É responsável pela configuração do CPU, da memória e do armazenamento das aplicações que correm nesse mesmo cluster, permitindo ainda que outras frameworks, para além do MapReduce, corram sobre o Hadoop, como o Spark ou o Flink [25]. A ideia fundamental do YARN é segmentar as funcionalidades de gestão de recursos e monitorização das tarefas em processos diferentes. Existe por

isso um ResourceManager global, e um ApplicationMaster por aplicação [26]. É possível observar esta estrutura na Figura 2.3. Esta arquitetura YARN é então constituída por 4 componentes principais:

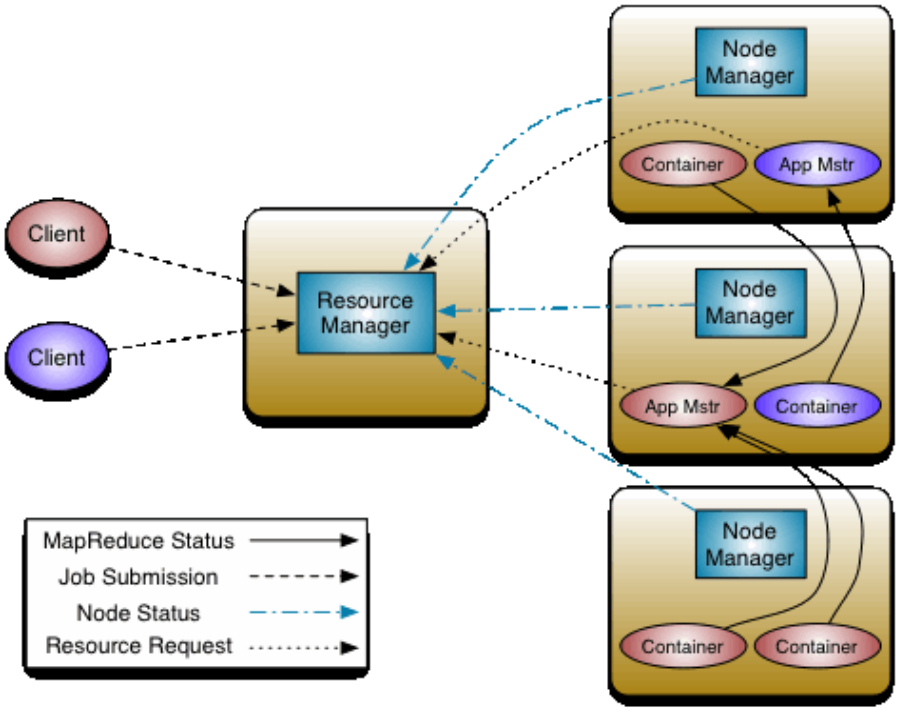


Figura 2.3: Arquitetura e Componentes Principais do Hadoop YARN

- Resource Manager (RM), que gere os recursos disponíveis no cluster, alocando-os para diferentes aplicações;
- Node Manager (NM), que está presente em todos os nós do cluster, e é responsável por monitorizar a utilização dos recursos disponíveis e comunicar ao Resource Manager, sendo constituído por 2 principais componentes: o Scheduler e o Applications Manager;
- Aplication Master (AM), que consiste numa instância associada a uma aplicação. Coordena a execução da aplicação, solicitando recursos ao RM e monitorizando as tarefas nos NMs;
- Container, a unidade fundamental de recursos alocada pelo YARN, e contituidos por um determinado conjunto de recursos físicos como CPU e memória [27].

Um fluxo normal de interações entre estas diferentes componentes durante uma execução de uma aplicação, pode ser observado na Figura 2.4. O RM, depois de receber o pedido do cliente (1), aloca um container para iniciar o AM (2), que se regista no RM (3) e com ele negoceia os containers necessários para a aplicação em questão (4), notificando depois o NM para estes serem iniciados (5). A aplicação corre depois nesse container (6). O cliente interage com o RM para monitorizar o estado da aplicação

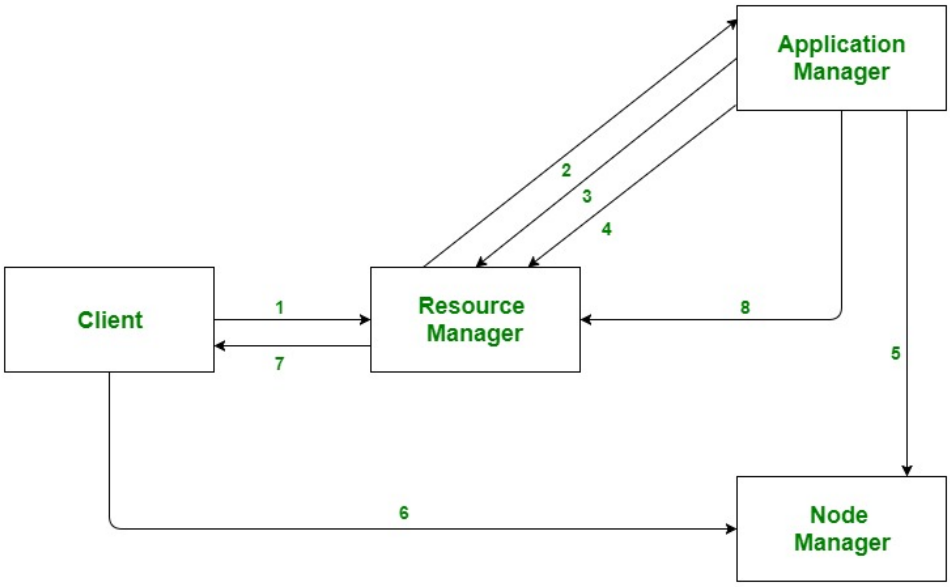


Figura 2.4: Fluxo normal numa aplicação sobre Hadoop YARN

(7) e, por fim, após a conclusão do processamento, o AM cancela o registo efetuado (8) no passo (3) no RM.

Para além das funcionalidades relacionadas com o armazenamento e a gestão dos dados distribuídos, oferece ainda capacidades de processamento distribuído, em particular o modelo MapReduce já mencionado e representado na Figura 2.5, que permite a divisão das tarefas de processamento em sub-tarefas menos complexas, que serão executadas de forma paralela pelos nós da rede. É portanto importante distinguir as 2 etapas fundamentais deste modelo de programação: a *Map*, onde os dados são processados e tranformados, e a *Reduce*, onde os resultados anteriores são combinados para produzir o *output* final. O utilizador define uma função de map a ser executada pelos diferentes nós consoante um input (chave, valor) (K, v), que irá também gerar um conjunto intermédio de pares (K, v) [28, 29], a serem futuramente recolhidos e agregados. Aprofundando, na fase inicial, ou Map, o conjunto de dados de entrada é particionado em blocos, tipicamente igual ou proporcional ao número de nós disponiveis. Cada nó excuta uma função de map que transforma os dados de entrada em pares (chave, valor) ou (K, v), que serão usados numa futura fase de agregação. Na fase Reduce, os pares (K, v) são agrupados ou agregados, considerando os pares enviados por cada nó. Este modelo apresenta também algumas limitações, particularmente a nível de latência em processos iterativos, como modelos de machine learning e outros que requiram múltiplas iterações, leituras e armazenamento sobre os dados iniciais e sobre dados intermediários. Assim, esta metodologia não parece ser a ideal para previsões de séríes temporais [30].

Além do Hadoop, outra ferramenta fundamental e popularmente utilizada no que diz respeito ao pro-

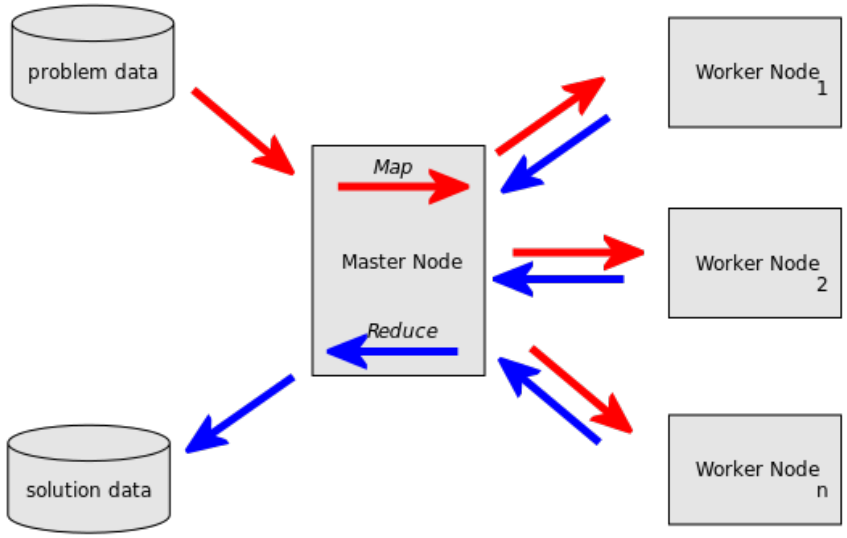


Figura 2.5: Arquitetura MapReduce

cessamento de grandes volumes de dados em tempo real (principalmente em aplicações iterativas e de processamento em tempo real), é o Apache Spark [31], uma *framework open-source* para computação distribuída que oferece uma interface para a execução de aplicações em clusters, oferecendo paralelismo absoluto. Supera algumas limitações do MapReduce e apresenta melhores performances e resultados mais rápidos [31], através de um modelo de processamento em memória (*in-memory*), sendo apropriado para aplicações que exigem baixas latências e boas velocidades de processamento, como algoritmos de machine learning, ou análises em tempo real. Este processamento em memória, reduz a necessidade de acessos ao disco, permitindo que os dados sejam armazenados em memória RAM durante as operações, reduzindo significativamente os tempos de execução. Suporta ainda APIs em diversas linguagens, como Python, Java ou R. Dentro da arquitetura Spark, é possível identificar 6 componentes fundamentais [32]:

- Spark Core Engine, o 'núcleo' da framework que gere os clusters, a execução de tarefas e a leitura e escrita de dados. Este Core introduz os RDD (*Resilient Distributed Datasets*) que servirão de alicerce para todas as operações, servindo para armazenar os dados em memória, e reduzindo assim significativamente os tempos de execução. Desenvolvido como uma extensão ao modelo MapReduce [21], estes RDDs permitirão que os dados sejam particionados e processados paralelamente pelos nós da rede, otimizando a utilização dos recursos disponíveis e reduzindo a

latência.

- Spark SQL, cujo nome origina no seu mecanismo de interação com os dados, semelhante ao do SQL, linguagem a qual também usa. Este módulo, implementado na camada acima do Core, é fundamental no funcionamento e otimização das consultas dos dados armazenados em RDDs ou DataFrames, criando de certa forma uma camada de abstração no acesso e interação com os dados distribuídos. Permite aos utilizadores um conjunto de operações sobre os dados, que podem estar e ser acedidos em diferentes formatos como Json, Hive ou JDBC.

- Spark Streaming, que possibilita o processamento de fluxos de dados em tempo real. Esta componente é responsável por dividir esse fluxo em pequenos lotes de dados (*micro-batches*) para análises em tempo real, um processo denominado de *micro-batching*.

- MLlib (*Machine Learning Library*), uma biblioteca de machine learning que disponilibiza variados algoritmos (como regressões logísticas e lineares, SVMs, classificadores Naive-Bayes, clusters K-Means, árvores de decisão...).

- GraphX, uma biblioteca para análise e processamento paralelo de grafos.

- SparkR, uma API que resolve um problema relevante da linguagem R, a sua limitação máxima de apenas 1 nó para processamento de dados, o que se traduz numa incapacidade em lidar eficientemente com grandes volumes de dados. Esta biblioteca do Apache Spark contorna essa limitação ao integrar o ambiente de programação R com a capacidade de processamento distribuído do Spark, utilizando o conceito de DataFrames distribuídos, e podendo ainda estar conjugado com algoritmos de machine learning da MLlib (como os já referidos) em ambientes distribuídos.

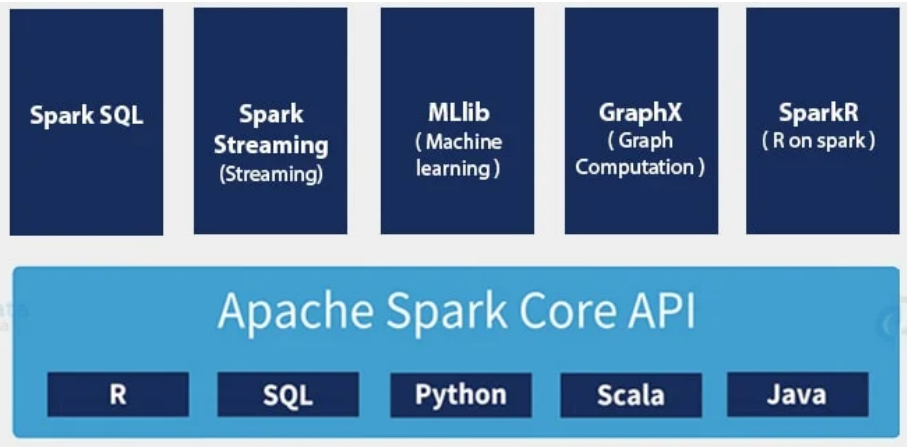


Figura 2.6: Arquitetura Spark

2.4 Sistemas de Previsão Meteorológica

As séries temporais desempenham um papel crucial em diversas áreas práticas, como a meteorologia ou a economia [33]. A previsão dessas mesmas séries temporais, baseia-se primeiramenter em análise de dados históricos, como um dataset constituído por observações meteorológicas, e, com base nos padrões observados, prever o seguimento dessa série num intervalo futuro. Pelo contrário, os modelos Numerical Weather Predictions (NWP) (*Numerical Weather Prediction*), e estando ilustrado na Figura 2.7 uma possível representação gráfica, que dependem fortemente da capacidade computacional dos executantes, baseiam-se em dados atmosféricos correntes, usando-os com input para produzir uma análise e prever condições meteorológicas futuras [33].

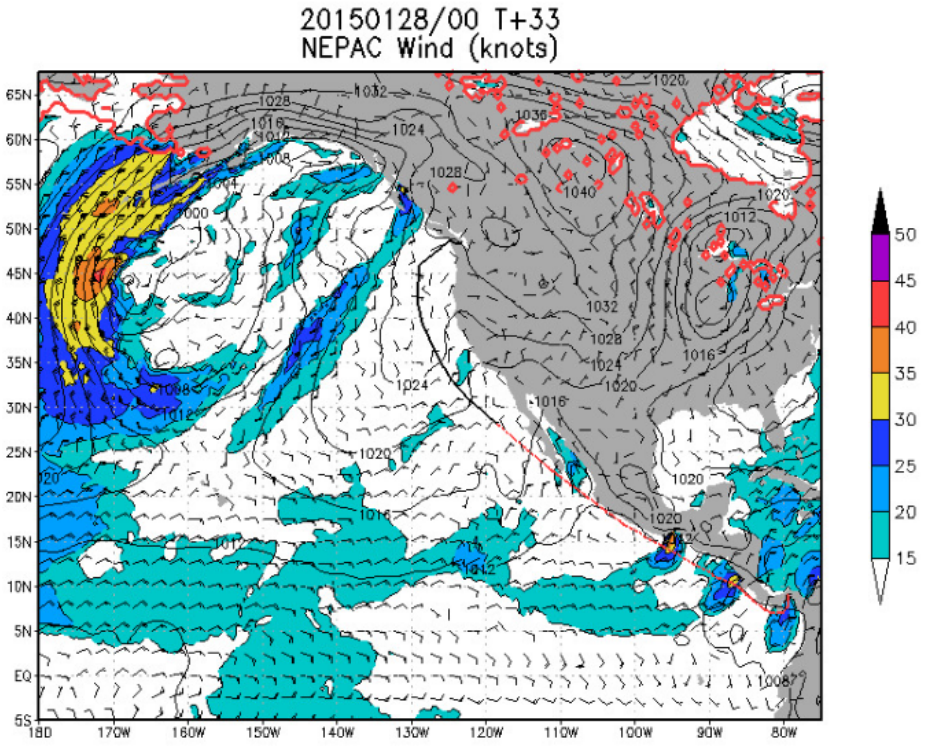


Figura 2.7: Modelo NWP

Outro campo relevante e muitas vezes associado à computação em edge, é o machine learning (ML), principalmente no contexto smart cities já mencionado. A combinação de tecnologias edge com machine learning é considerada muito promissora na área do IoT [3]. Considerando os enormes fluxos de dados permanentes que um contexto deste tipo gera, se as tecnologias em ação in-loco, para além das features que já referimos, forem também capazes de se adaptarem e aprenderem com esses fluxos em tempo real, o potencial torna-se ainda maior. Este tipo de modelos oferece uma abordagem mais flexível para modelagem de séries temporais, sendo capazes de capturar padrões não lineares

e variações complexas. Algumas técnicas populares são redes neurais recorrentes, incluindo LSTMs (*Long Short-Term Memory*) e GRUs (*Gated Recurrent Units*), que foram projetadas para lidar com dados sequenciais, como séries temporais. Estas redes, são compostas por camadas interconectadas de neurónios artificiais que processam e transmitem dados. Como é possível observar no diagrama da Figura 2.8, a camada de input (input layer) recebe os dados brutos diretamente do conjunto de dados de treino. Existe depois uma ou mais camadas ”ocultas”que aplicam transformações e permitem que a rede aprenda e identifique padrões complexos e não-lineares. O número de camadas pode variar em função da complexidade do problema. A camada final é responsável por produzir o resultado final do modelo, seja uma previsão ou classificações. Entre outras abordagens possíveis, destacam-se também os modelos baseados em árvores de decisão, ou *decision trees*, como o Random Forest Classifier [34], o Decision Tree Classifier ou o Extra Tree [35]. A um nível ainda mais complexo, temos modelos de deep learning como redes neurais profundas, que permitem identificar padrões espaciais e temporais em grandes volumes de dados [36]. Essas redes profundas possuem múltiplas camadas ocultas de forma a serem capazes de capturar relações mais sofisticadas.

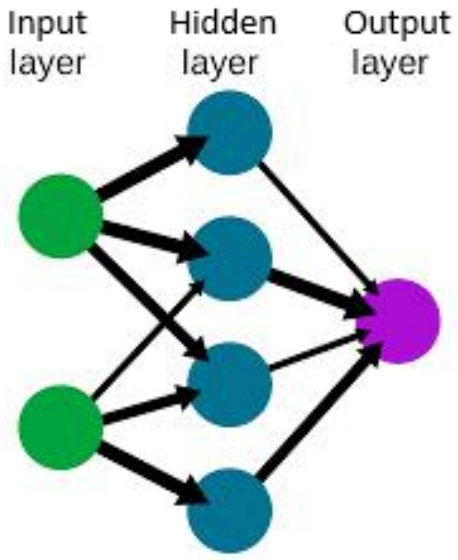


Figura 2.8: Estrutura base de uma rede neural

Voltando aos sistemas de modelagem estatística tradicional, que assumem que a série temporal segue certos padrões como linearidade, estacionaridade ou sazonalidade: um popular e tradicional algoritmo de modelagem estatística e de previsão de séries temporais é o modelo Auto Regressive Integrated Moving Average, ou AutoRegressive Integrated Moving Average (ARIMA) [37]. É um modelo fácil de adoptar pois não requer tanto esforço computacional como modelos de Machine e Deep Learning, como as redes neurais convolucionais (Convolutional Neural Network (CNN)) ou as Recurrent

Neural Network (RNN), algo que se pode revelar bastante útil em função do tipo de dispositivos na edge, possibilitando, por exemplo, que até os próprios sensores responsáveis pelas medições possam ter capacidade de executar modelos deste tipo [38]. Baseia-se na assunção que *a série temporal é linear e segue uma distribuilção estatística, como uma distribuição normal* [33]. Mais apropriado ainda a previsões de séries identificadas como sazonais, é o Seasonal Auto Regressive Integrated Moving Average, ou Seasonal AutoRegressive Integrated Moving Average (SARIMA), e que é formado ao adicionar os parâmetros sazonais ao input do modelo ARIMA:

$$\text{ARIMA (p, d, q) (P, D, Q)}_m$$

Sendo (p, d, q) os parâmetros não-sazonais, e (P,D,Q) os sazonais, onde *m* é a periodicidade da sazonalidade identificada. A definição dos parâmetros (p, d, q) e (P, D, Q) irá ser explorada e demonstrada no capítulo 5. Relativamente ao período da sazonalidade, se, por exemplo, tivermos registos ou dados mensais (de temperaturas ou precipitação, consideremos), e estes apresentarem padrões anuais, então $m = 12$.

Peng Chen et al. [33], em 2018, exploraram técnicas de previsão de séries temporais, neste caso a temperatura do ar, com modelação SARIMA, e demonstram os seus bons resultados a nível de precisão, usando dados históricos de temperaturas médias mensais em Nanjing, na China, desde 1951 até 2014 para treinar e parametrizar o modelo, e usando os dados de 2015 a 2017 para o testar. O processo de seleção do modelo foi determinado pelo Akaike Information Criterion (AIC), uma métrica amplamente utilizada para comparar modelos estatísticos, e também ela implementada na função de parametrização 'auto-sarima'. Os resultados do modelo foram avaliados com métricas como o erro médio quadrático (MSE), tendo sido concluído que o SARIMA é um modelo robusto e confiável na previsão de série temporais sazonais, particularmente em contextos meteorológicos, e que as técnicas de modelagem estatística podem ser aplicadas a dados e reais e produzir informações e análises valiosas.

Meenal et al. [34], em 2021, apresentaram um modelo de previsão meteorológica baseado no algoritmo Random Forest (RF). O estudo focava-se na previsão de radiação solar e velocidade do vento em Tamil Nadu, na Índia, utilizando dados como temperatura máxima e mínima, pressão atmosférica, humidade, e posicionamento geográfico. Os testes foram realizados dividindo os dados em 70% para efeitos de treino, e 30% para testes. Os resultados indicaram que o modelo Random Forest superou outros como o SVM (Support Vector Machine), apresentando menor erro médio quadrático (MSE) e um coeficiente de determinação superior (R^2). Por isso, o estudo acaba por rejeitar a necessidade de equipamentos de medição mais complexos e potencialmente dispendiosos para fazer este tipo de capturas e previsões.

Já Ajina et al. [36], em 2023, optaram por desenvolver um sistema de previsões meteorológicas baseado em redes neurais artificiais, ou Artificial Neural Network (ANN). Foram utilizados conjuntos de dados que incluíam diversas variáveis, como temperatura máxima e mínima, pressão atmosférica, velocidade do vento e umidade, sendo que o treino e a precisão do modelo foram avaliados através da métrica de *mean square error*. Foi então concluído que as ANN têm capacidade de oferecer previsões confiáveis para dados climáticos complexos e com padrões não lineares, reforçando o potencial das ANN em substituir métodos tradicionais de previsão, onde a precisão e a complexidade dos dados são elementos críticos.

Análise e Discussão

Em síntese, os trabalhos analisados evidenciam o potencial das abordagens baseadas em séries temporais e em algoritmos de *machine learning* para previsões meteorológicas, apresentando resultados satisfatórios em termos de precisão. Contudo, observa-se que a maioria das soluções estudadas depende fortemente de infraestruturas cloud centralizadas ou de dispositivos dedicados com elevada capacidade computacional, o que limita a sua aplicabilidade em contextos dinâmicos e com menores recursos, como os de uma smart city. Assim, para o nosso contexto de análise, torna-se pertinente explorar arquiteturas fog e edge, combinando mecanismos de processamento paralelo (como MapReduce, utilizado na fase de parametrização dos modelos) e MapCollect (na fase de previsão), com capacidades de executar workloads de previsão meteorológica de forma distribuída e próxima das fontes de dados, reduzindo a latência e a dependência de comunicações com a cloud.

3

Solução

Conteúdo

No seguimento de reflexões e análise sobre este cenário de smart cities em edge, visando encontrar soluções que ajudem na evolução e implementação desse conceito, esboçou-se uma possível abordagem para o desenvolvimento de uma plataforma, que possibilita ao utilizador, através dos dispositivos que este possua no momento, e tirando partido dos seus recursos computacionais (e também dos recursos daqueles à sua volta e que tenham capacidade para auxiliar), realizar previsões meteorológicas naquele local, para os próximos dias. A aplicação assente nessa plataforma, ofereceria 3 intervalos de previsão: 1 dia, 3 dias e 1 semana, escalando também a quantidade de dados necessários para o cálculo de cada uma dessas previsões. Em termos das variáveis a serem previstas, o sistema ofereceria variáveis distintas, que possam ser selecionadas indidivualmente, ou em conjunto, de forma a oferecer flexibilidade na carga de trabalho computacional em função das informações pretendidas. Um protótipo de um *frontend* possível para ecrã inicial da nossa aplicação está demonstrado na Figura 3.1, cujo código desenvolvido através de bibliotecas 'streamlit', para Python, dessa, e das seguintes páginas, está presente em Listagem A.4.

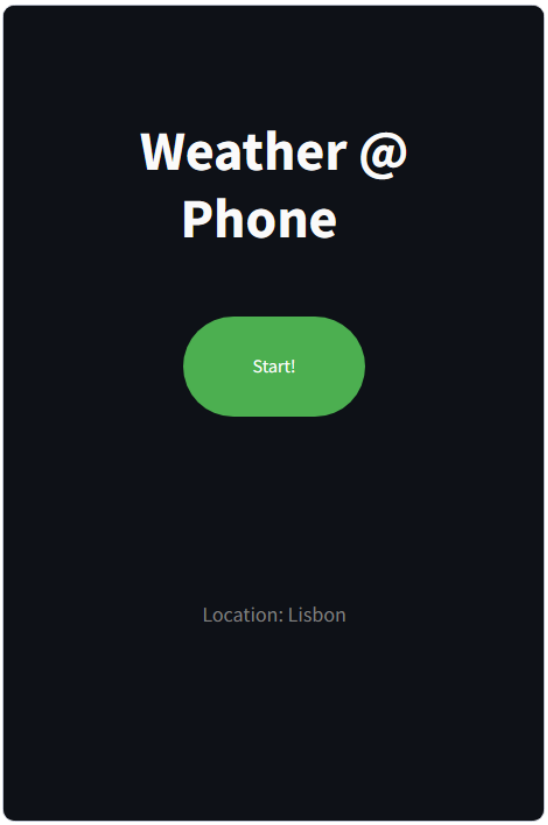


Figura 3.1: Possível ecrã inicial da plataforma

A plataforma foi desenvolvida em Python para dispositivos com conectividade e capazes de executar modelos de previsão lineares. Um possível ecrã de parâmetros de interesse a selecionar pelo utilizador, está demonstrado na figura 3.2(a), sendo que a página demonstrativa dos resultados da previsão, está exemplificado na figura 3.2(b).

O trabalho computacional exigido por esses modelos, irá ocorrer sobre dados históricos relativos ao local em questão, e é portanto necessária a obtenção dos dados essenciais à previsão (seja essa recolha feita localmente, ou através de pedidos API ou bibliotecas disponíveis que o permitam - a Meteostat, por exemplo, oferece uma biblioteca em Python que permite aceder a uma vasta base de dados históricos relativos a inúmeros locais). Independentemente de como é realizada a recolha dos mesmos por parte da camada *fog*, esta poderá ainda atualizar periodicamente um sistema de ficheiros partilhado ao qual os nós da camada *edge* terão depois acesso, não havendo necessidade de tráfego destes dados entre essas camadas, ou distribuição feita via HDFS, permitindo aos nós atuarem sobre dados locais. Estes dados são alvo de um tratamento por parte do dispositivo móvel do utilizador (telemóvel, tablet, laptop, . . .). O utilizador mobile utiliza os seus próprios recursos computacionais para efetuar cálculos sobre os dados obtidos. Este tratamento pode hipoteticamente resultar em análises estatísticas, previsões ou decisões. Esta plataforma irá incidir sobre previsões horárias de variáveis

(a) Formulário **(b)** Resultados

Figura 3.2: User Interface completa

meteorológicas. No nosso caso, esta camada fog poderá ser constituída pelos Access Points (APs) disponíveis que funcionariam como ”local servers”(ou LS), E que irão oferecer: dados históricos relevantes para o local em que se encontra - um possível fluxo de dados entre os AP/LS e os nós será demonstrado a seguir, na Figura 3.4; garantias de execução em caso de indisponibilidade pelo nó em questão, ou pelos nós das redondezas; e, por fim, auxílio no aumento da qualidade dos resultados apresentados, através da optimização cooperativa dos parâmetros dos modelos de previsão. O local server (LS), presente na camada fog, faz pedidos periódicos para manter os dados necessários aos nós atualizados e armazenados próximos dos mesmos. Sempre que um dispositivo (nó) novo entra no cluster de um LS/AP, terá de o notificar. Podemos por isso definir que todos os nós do cluster que estejam disponíveis para contribuir com recursos computacionais, terão de enviar mensagens periódicas (ou *heartbeats*) ao respetivo LS/AP, tendo este sempre conhecimento por quais nós pode distribuir as tarefas.

Tendo essa informação, e assumindo que a parametrização dos modelos é feita de forma periódica e distribuída, a camada fog pode agora segmentar e distribuir essa tarefa. Introduzindo um contexto

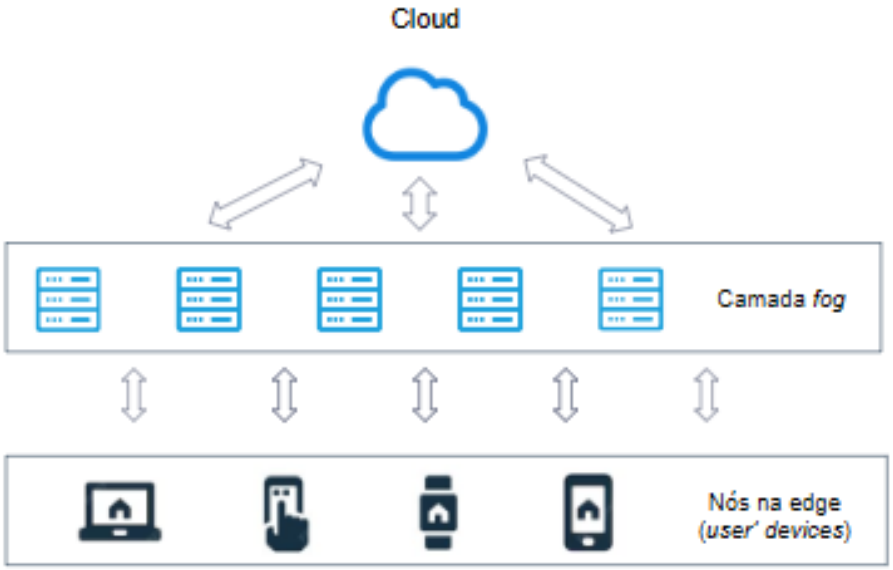


Figura 3.3: Arquitetura da plataforma

aplicacional, considerando agora que um utilizador quer realizar uma previsão, este terá de selecionar o intervalo pretendido e as variáveis de interesse. Esse pedido é entregue ao respetivo acess point (AP), que irá agora segmentar a carga da execução do modelo de previsão, de forma a distribui-la pelos nós locais da rede capazes de participar. Por fim, recolhe os resultados dos nós participantes, e entrega o resultado da previsão ao utilizador.

3.1 Modelo de Previsão

Para início de desenvolvimento da plataforma, e com base na investigação realizada anteriormente, começámos por implementar o modelo ARIMA nos nós locais, realizando de seguida testes para avaliar a sua eficiência (recorrendo a métricas de avaliação explicadas mais à frente, no capítulo 5, secção 5.1.

O primeiro passo seria então a definição dos valores dos parâmetros a utilizar neste modelo: 'p, d, q'. Foi utilizado um *dataset* de registos históricos de temperaturas em Lisboa para o ano de 2023, obtido através da base de dados da Meteostat. Começando pelo parâmetro 'p' (ou *lag order*), ou seja, a ordem do modelo auto-regressivo, realizaram-se testes de Partial Correlation (Partial Autocorrelation Function (PACF)) para determinar o número de lags significativos existentes até estabilizar no 0, cujos resultados podem ser observados na figura 3.5. Podemos então identificar cerca de 25 lags significativos, pelo que será essa a nossa *lag order*.

Passamos de seguida para a análise do parâmetro d, referente ao grau de diferenciação da série. Uma das formas para determinar esse parâmetro, é realizar o teste Augmented Dickey-Fuller (ADF), de

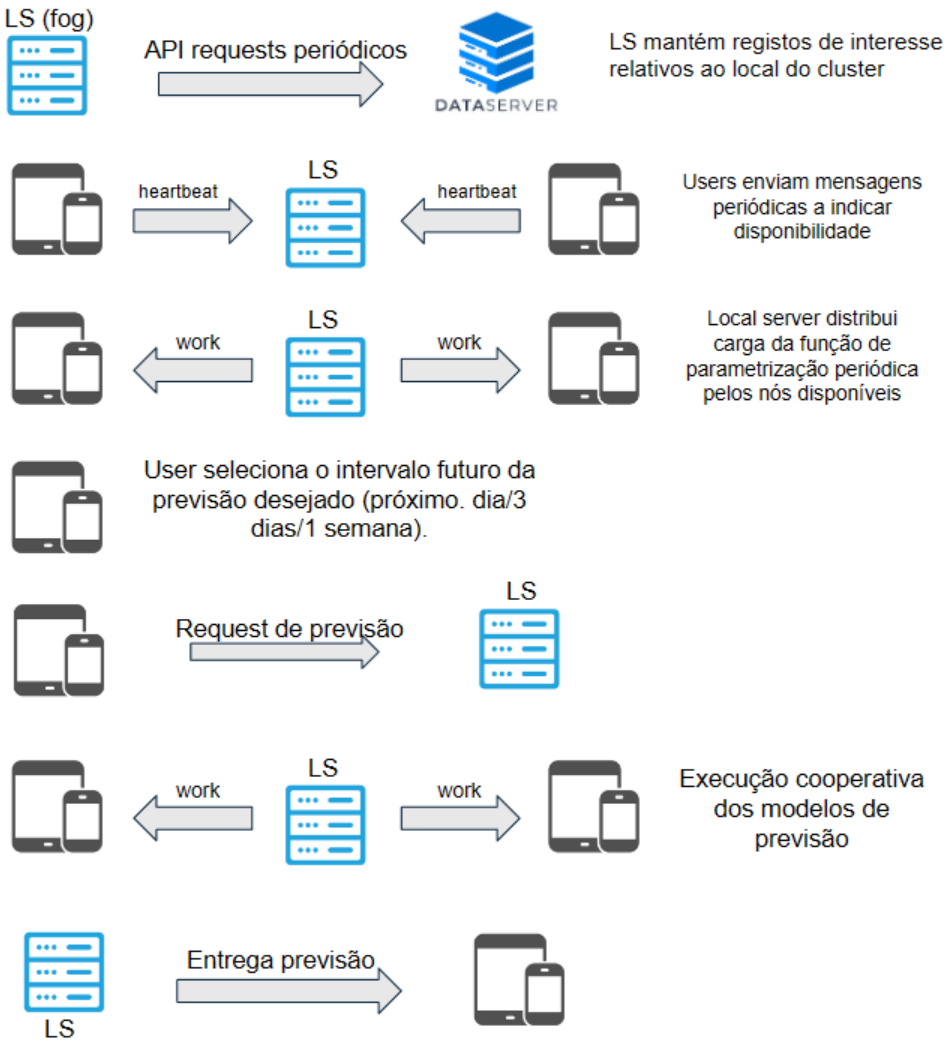


Figura 3.4: Fluxo de dados da plataforma

forma a verificar se a série é estacionária ou não. Após a execução desse teste, verificou-se que

ADF Statistic: -3.4494898186886993

p-value: 0.00938310848841729

O que significa, como o *p-value* é menor que 0.05 e o ADF estatístico é consideravelmente negativo, podemos concluir que se trata duma série não-estacionária, não havendo necessidade de diferenciação. Assim, iremos definir o parâmetro d como d = 0.

Prosseguindo então para o terceiro e último parâmetro dado para execução de um modelo ARIMA, o 'q', ou a ordem do modelo de média móvel. Realizando um teste de Autocorrelation Function (ACF) (Autocorrelation Function), e procedendo à análise do gráfico resultante (Figura 3.6), que consiste essencialmente na identificação do número de lags necessários até a oscilação decair para zero. Como

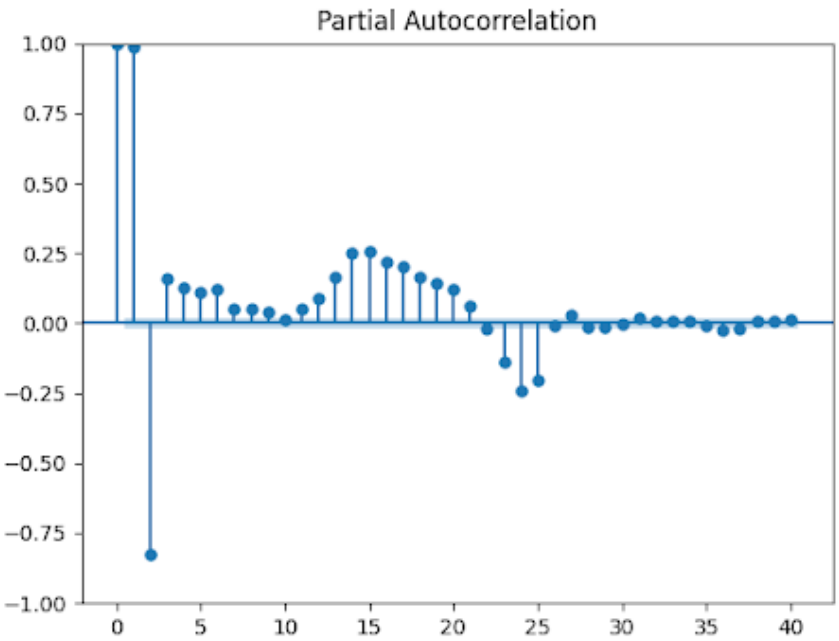


Figura 3.5: Teste PACF

podemos então observar no nosso gráfico da Figura 3.6, mesmo alargando o scope do teste até um número alto de lags (100), o nosso valor de ACF oscila entre 1 e 0.5 sem nunca convergir para zero. O que significa então este resultado e como resolver este obstáculo? O facto do nosso gráfico de ACF nunca convergir para zero num número de lags considerado razoável, é um forte indicador de sazonalidade no nosso dataset. Isso significa que um modelo ARIMA simples (não sazonal) pode não capturar corretamente a dinâmica dos dados. Tendo isso em conta, vamos então investigar o modelo ARIMA sazonal, o SARIMA, comparando resultados entre os 2 modelos, com parâmetros otimizados para SARIMA através de funções de parametrização, como o grid search, e utilizando os seguintes intervalos possíveis (assumindo o parâmetro S de sazonalidade como 24, devido ao padrão sazonal diário da série, com período de 24, o que reflete a periodicidade horária): paramGrid = {'p': [0, 1, 2], 'd': [0, 1], 'q': [0, 1, 2], 'P': [0, 1, 2], 'D': [0, 1], 'Q': [0, 1, 2]} e com parâmetros ARIMA (data, order=(25, 0, 0) selecionados anteriormente, e fazendo recurso de algumas das métricas de avaliação da precisão de modelos de previsão aprofundadas no capítulo 5. A nossa função de *grid search* selecionou como parâmetros ideais para o modelo SARIMA executado sobre o conjunto de dados providenciado: (order=(1, 1, 1), seasonalOrder=(1, 1, 1, 24)).

Por vezes, a interpretação gráfica dos resultados - apesar de poder adiantar algumas pistas - não traduz por completo a eficiência de cada modelo, pelo que vamos usar algumas métricas para nos auxiliar na comparação de resultados entre um modelo ARIMA e SARIMA sobre esse mesmo conjunto

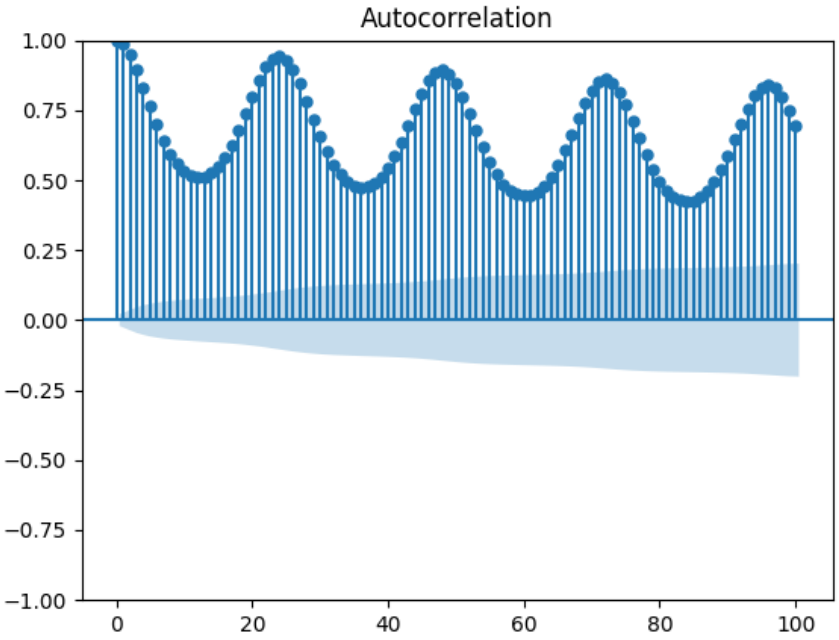


Figura 3.6: Teste ACF

de dados:

ARIMA

MSE: 2.025836642261745 — RMSE: 1.4233188828445105 — MAE: 1.2107339165189261

SARIMA

MSE: 1.8698859231700808 — RMSE: 1.367437721861614 — MAE: 1.1585487266301202

Fazendo recurso das métricas MSE (que mede a média dos quadrados dos erros - a diferença entre a previsão e o valor real portanto) explanadas em detalhe no capítulo 5 - e pelo que se esta métrica define um valor de erro, significa que quanto menor o seu valor, melhor será o modelo em questão - e também da métrica de RMSE, e comparando os resultados para cada um dos modelos, é possível concluir que o modelo SARIMA apresentou um desempenho ligeiramente superior, conclusão que é corroborada também pelos valores de RMSE, expressados nas mesmas unidades dos dados originais (ou seja, a raiz do quadrado do erro médio na previsão das temperaturas para o modelo ARIMA foi cerca de 1.42 ºC, enquanto que para o modelo SARIMA foi cerca de 1.37 ºC, um resultado ligeiramente melhor no ajuste aos dados, apesar da diferença ser reduzida).

É possível também verificar visualmente, nos gráficos de previsão dos modelos presentes na Figura 3.7, as previsões efetuadas por ambos os modelos, em Lisboa, para um intervalo de 3 dias, iniciado à meia-noite de 28 de dezembro de 2023, e findado às 23:59h de dia 31 de dezembro. Para treino

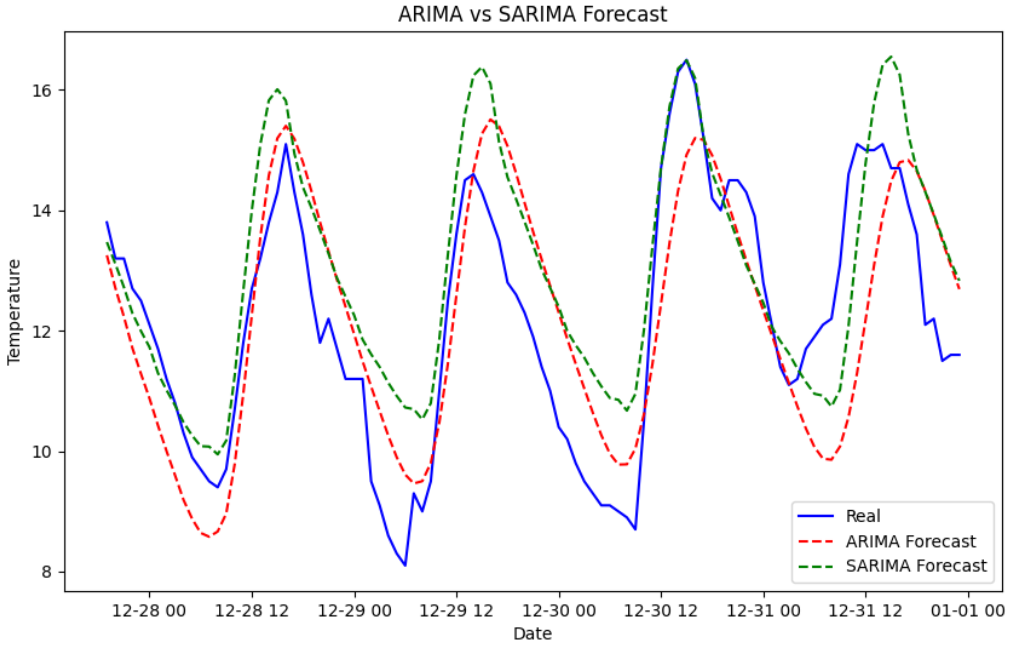


Figura 3.7: Comparação de resultados reais observados com previsões ARIMA e SARIMA

dos modelos, foram utilizados os dados registados em 2023, desde a primeira hora do dia 1 de janeiro, até às 23:59h de 28 de dezembro desse ano. A linha azul representa as observações reais nesse intervalo de 3 dias, e como é possível verificar, é difícil analisar e distinguir a qualidade dos modelos apenas observando visualmente as suas previsões, sendo por isso necessário recorrer às métricas mencionadas.

Findas as avaliações dos modelos de previsão, e considerando fatores como a simplicidade dos modelos para maximizar a possibilidade destes serem executados por dispositivos *edge* e próximos das fontes de dados (como sensores de temperatura, humidade ou precipitação), concluiu-se que a melhor opção para efetuar as nossas previsões meteorológicas seria os modelos SARIMA.

3.2 Arquitetura Funcional

Podendo então distinguir dois papéis claros distintos dentro da nossa estrutura, os nós da camada fog, que servirão de masters/orquestradores, e os nós da camada edge, que por sua vez irão atuar como workers/escravos, há que planificar as responsabilidades e tarefas de cada um, e de que forma a interação entre os mesmos decorrerá. Conhecendo já o fluxo da plataforma, demonstrado na fig. 3.4, podemos delinear as seguintes possíveis 'responsabilidades' associadas aos nós fog:

- Receber/obter os dados

- Armazenar e processar localmente
- Orquestrar o offloading de tarefas, por exemplo, delegando parametrizações ou previsões parciais a dispositivos móveis
- Responder a pedidos dos nós edge por previsões meteorológicas
- Atuar como ponto de decisão para geração de alertas (temperatura ou velocidade do vento excederem determinados thresholds, por exemplo)

Já os nós locais da camada edge, ficam encarregues de informar periodicamente da sua disponibilidade (por broadcast de um heartbeat para a rede) ao respetivo nó a ele associado da camada superior. Assim, o nó fog poderá atualizar o seu inventário de workers e distribuir por eles as tarefas necessárias, seja a execução de funções de parametrização ou a execução dos modelos de previsão.

Independentemente dos nós *workers* disponíveis no cluster, o nó *master* funcionará também como *worker* em simultâneo. Desta forma, poderá realizar a necessária parametrização periódica do modelo de previsão, e estará apto a executar esse modelo de forma rápida assim que algum utilizador o solicitar. Esta intenção deriva do demorado tempo de execução deste tipo de funções de parametrização, que obrigaria, como demonstraremos mais à frente, a esperas superiores a 1 minuto para previsões simples de variáveis singulares, algo que contraria os objetivos de eficiência desta arquitetura de plataforma.

4

Desenvolvimento da Proposta

Conteúdo

Este capítulo apresenta em detalhe o processo de desenvolvimento da plataforma proposta, centrada na aplicação do paradigma fog computing para a realização de previsões meteorológicas em dispositivos localizados na extremidade da rede (edge). A proposta visa criar um sistema distribuído e escalável, capaz de realizar o processamento e a análise de dados climatéricos em tempo real, o mais próximo possível da origem dos dados, e de forma totalmente distribuída. Esta abordagem permite não só reduzir a latência, o tráfego de dados e a carga nos centros de dados centrais, como também capacitar os nós locais com autonomia para gerar alertas e obter dados de interesse nem necessidade de ligação a servidores centrais e redes externas. A plataforma foi desenhada para operar num cenário representativo de cidades inteligentes (smart cities), em que sensores e dispositivos móveis recolhem continuamente informações do ambiente — como temperatura, humidade e pressão atmosférica — e as transmitem para nós intermédios (fog nodes) com capacidade de processamento. A partir destes dados, podem ser realizadas análises preditivas com o objetivo de antecipar condições meteorológicas relevantes, bem como detetar situações anómalas que possam indicar, por exemplo, risco de rajadas

fortes, temperaturas extremas ou outras condições com relevo de alerta. Para tornar esta proposta funcional, recorreu-se a um conjunto de ferramentas de código aberto, amplamente utilizadas em contextos de big data e computação distribuída. A componente de armazenamento distribuído do Apache Hadoop foi utilizado como base para o armazenamento distribuído de dados, através do seu sistema de ficheiros HDFS (Hadoop Distributed File System), que permite o acesso local (ou remoto, acessando aos DataNodes) em simultâneo e tolerante a falhas por parte de múltiplos nós. Já o Apache Spark foi a tecnologia escolhida para o processamento em memória de grandes volumes de dados de forma distribuída, beneficiando da sua eficiência na execução de tarefas paralelas e da sua flexibilidade para integrar modelos de machine learning e séries temporais. As secções seguintes deste capítulo descrevem, de forma estruturada, as etapas envolvidas na criação do ambiente de execução da plataforma, incluindo a instalação e configuração de cada componente, a definição da arquitetura do cluster e a realização de testes de funcionamento. São ainda discutidos os principais desafios técnicos enfrentados durante o processo e as estratégias adotadas para os ultrapassar, contribuindo para a construção de uma solução robusta, modular e preparada para evoluir com base em novos requisitos e contextos de aplicação.

4.1 Configuração do cluster

Tendo já definido que papéis são necessários ser desempenhados dentro da nossa arquitetura, podemos agora delinear mais concretamente aquilo que suportará a estrutura da plataforma. A construção de um ambiente distribuído funcional é, obviamente, um dos pilares essenciais para a execução eficiente da proposta apresentada. Assim, neste capítulo, descreve-se o processo de instalação e configuração do cluster que servirá de base à plataforma fog idealizada. Este cluster permite tanto o armazenamento distribuído de grandes volumes de dados como o seu processamento paralelo em tempo real, suportando os diversos serviços necessários ao funcionamento da arquitetura.

Foram utilizados componentes amplamente reconhecidos no ecossistema de Big Data, nomeadamente o Apache Hadoop e o Apache Spark. O Hadoop, com o seu sistema de ficheiros distribuído HDFS, permite garantir o acesso e a replicação de dados entre múltiplos nós. Já o Spark é responsável pelo processamento eficiente dos dados, tirando partido de um modelo de execução em memória e de suporte nativo a tarefas paralelas e algoritmos preditivos. Apesar do Apache Hadoop incluir um motor de processamento distribuído próprio, também baseado no paradigma MapReduce, optou-se por utilizar o Hadoop exclusivamente como sistema de armazenamento distribuído (HDFS), delegando o processamento de dados ao Apache Spark. Esta decisão foi motivada por várias razões: em primeiro lugar, o modelo de processamento do Spark é consideravelmente mais eficiente e moderno, permitindo a execução de tarefas em memória (*in-memory computation*), o que reduz significativamente o

tempo de execução, especialmente em workloads iterativos como algoritmos de séries temporais ou de machine learning. Além disso, o Spark disponibiliza interfaces de alto nível, como Spark SQL, DataFrames e integração nativa com bibliotecas de ciência de dados, tornando o desenvolvimento mais ágil e flexível [31]. Como já referido durante a fase de investigação na seção 2.3, o motor de processamento MapReduce do Hadoop apresenta uma abordagem mais rígida e baseada em disco, o que resulta em maior latência, tornando-o menos adequado para sistemas que exigem baixa resposta e processamento em tempo real. Assim, o HDFS foi mantido como infraestrutura de suporte ao armazenamento distribuído, aproveitando a sua robustez, tolerância a falhas e compatibilidade com o ecossistema Spark.

Tabela 4.1: Resumo das tecnologias utilizadas e respetivos papéis na plataforma

Tecnologia	Papel no sistema	Vantagens principais
Apache Hadoop	Armazenamento distribuído	Sistema de ficheiros HDFS, tolerância a falhas, fiabilidade, compatibilidade com o ecossistema Big Data.
Apache Spark	Processamento distribuído	Execução em memória (in-memory), flexibilidade na programação, elevada velocidade de processamento, suporte a múltiplas bibliotecas (MLlib, SQL, streaming).

Cada uma destas ferramentas foi configurada num ambiente Linux baseado em Ubuntu 20.04.1, com os serviços organizados em máquinas virtuais separadas, representando diferentes nós da arquitetura. Foram também validadas as interfaces de monitorização disponibilizadas por cada framework, de forma a assegurar o correto funcionamento e comunicação entre os nós do cluster.

As secções seguintes detalham, passo a passo, o processo de instalação, configuração e validação de ambos os componentes.

4.1.1 Apache Hadoop

Para dar início à criação e configuração do ambiente em cluster, foi instalada a framework Apache Hadoop, anteriormente referida, numa máquina Ubuntu que servirá de Master a passos futuros, com o objetivo de aproveitar as capacidades do seu sistema de ficheiros distribuído, o HDFS. Este sistema, como já referido, permitirá o acesso partilhado a conjuntos de dados armazenados de forma distribuída, facilitando a atuação coordenada dos nós worker da rede. Foi então criado e configurado um cluster Hadoop funcional para aferir o funcionamento correto do HDFS, neste caso em single-node a funcionar como master e worker em simultâneo, de forma a confirmar o seu funcionamento.

Após criação do user 'hadoop', foi gerada a chave pública ssh sem password (`-P ''`) com o comando `ssh-keygen -t rsa -P '' -f /.ssh/id_rsa`, e guardada essa chave como authorized_key no diretório ssh: `cat /.ssh/id_rsa.pub >> /.ssh/authorized_keys`, tendo-se de seguida testado

o acesso ssh ao localhost. Futuramente, estas chaves terão de ser entregues aos nós worker do cluster.

Após instalação do software necessário (jdk, hadoop e ssh) na nossa máquina, e configuradas as variáveis de ambiente em bashrc, seguia-se a configuração dos ficheiros de propriedades do cluster: core-site.xml, hdfs-site.xml, mapred-site.xml e yarn-site.xml. Iniciou-se então o Hadoop Cluster, primeiramente o NameNode e o Datanode: `./start-dfs.sh`

Como podemos aferir na interface gráfica presente na Figura 4.1, o Datanode incializado encontra-se disponível e acessível por parte do worker node (192.168.218.139, com *system name* 'ubuntu'). E

Figura 4.1: Validação do acesso ao DataNode a partir de um nó Worker

de seguida, o YARN e NodeManagers com o comando `./start-yarn.sh`.

Para verificar a criação correta do cluster, acedeu-se ao http://localhost:9870 (NameNode UI), aos DataNodes também acessíveis através do mesmo endereço, e o YARN ResourceManager localizado em http://localhost:8088.

4.1.2 Apache Spark

Para a instalação do Apache Spark (versão 3.5.5), foi necessário, num primeiro momento, garantir a instalação das suas dependências principais, nomeadamente o Scala e o Git. Estes componentes são fundamentais, uma vez que o Spark é desenvolvido em Scala, e o Git facilita a obtenção de versões e repositórios associados ao projeto. Concluída esta etapa, procedeu-se à instalação do Apache Spark propriamente dito. Após a extração e colocação do diretório do Spark no sistema, foram configuradas as variáveis de ambiente necessárias, como SPARK_HOME e a atualização da variável PATH, de forma a permitir a execução dos comandos do Spark a partir de qualquer local no sistema.

Com a instalação concluída, foi iniciada a configuração do cluster Spark. Para isso, recorreu-se aos scripts disponibilizados pela própria framework. O nó mestre foi iniciado através do comando

`start-master.sh`, ficando disponível através da interface web no endereço http://192.168.218.140:8080/, onde é possível monitorizar o estado do cluster, os nós ativos e as tarefas em execução.

De forma a possibilitar o acesso HDFS aos nós worker, terão de ser entregues as chaves SSH aos elementos constituintes do cluster. Ou seja, assim que o nó master recebe o 1º heartbeat de um novo nó no cluster, terá de executar o comando `ssh-copy-id new_node`. De seguida, foram lançados dois nós worker associados ao master, utilizando o comando `start-worker.sh spark://master:7077`, onde 'master' corresponde ao hostname ou IP do nó mestre. Esta ligação permite que os workers se registem e passem a integrar o cluster, disponibilizando os seus recursos computacionais para as tarefas submetidas.

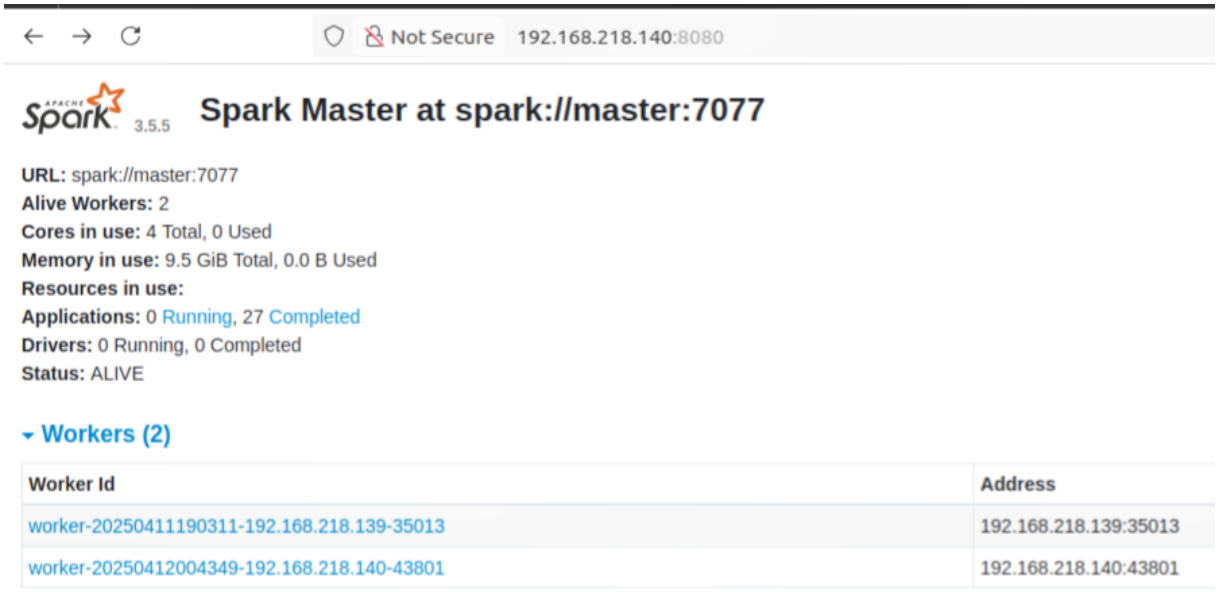


Figura 4.2: Estado dos nós Worker após ligação ao cluster Spark

A Figura 4.2 e a Figura 4.3 ilustram tanto a interface gráfica do Spark após a associação de dois nós worker ao cluster, como também mostra a sua correta configuração e o estado de disponibilidade dos recursos. Além disso, foram executados scripts simples de processamento distribuído, com o objetivo de confirmar a comunicação eficaz entre os componentes do cluster e a disponibilidade dos recursos computacionais.

Address	State	Cores	Memory
192.168.218.139:35013	ALIVE	2 (0 Used)	2.8 GiB (0.0 B Used)
192.168.218.140:43801	ALIVE	2 (0 Used)	6.7 GiB (0.0 B Used)

Figura 4.3: Visualização do cluster Spark após adição de workers

Nesse sentido, ao correr o comando `jps` (Java Virtual Machine process status) num nó worker para ver as instâncias JVM a decorrer nessa máquina, de forma a verificar a sua correta configuração, dever-se-á verificar tanto o processo 'Worker' relativo ao seu papel no cluster Spark, tal como o acesso aos ficheiros em HDFS, através do processo 'Datanode', como se pode observar na fig. 4.4.

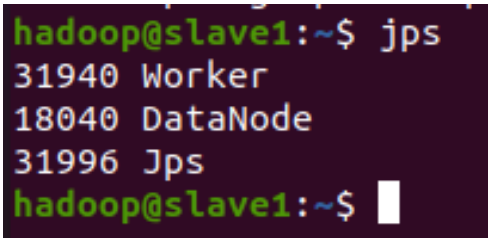


Figura 4.4: Processos Java ativos

4.2 Desenvolvimento de código para testes

Para testar as premissas de eficiência da nossa arquitetura, é então agora necessário correr uma aplicação que assente na execução de modelos de previsão simples, dentro do nosso cluster, e comprovar que, de facto, para além da redução de tráfego de dados para redes externas, existe também um acréscimo na eficiência da nossa plataforma distribuída, em oposição a uma execução que não envolva distribuição de tarefas e dados, e o tratamento em paralelo dos mesmos. Num sistema de previsões meteorológicas, que depende e recai num modelo de previsão, linear neste caso, o primeiro desafio a encontrar será a definição dos melhores parâmetros do mesmo, em função de um determinado dataset, como já foi discutido e analisado anteriormente. Tendo o modelo SARIMA destacado-se como a melhor opção neste contexto, e sabendo que a seleção dos melhores parâmetros pode variar consoante o conjunto de dados de treino, decidiu-se que seria necessária a execução periódica (diariamente por exemplo) de uma função de parametrização deste modelo. Sabendo da complexidade e carga computacional deste tipo de tarefa, perfila-se desde logo como um 'alvo' ideal para aplicar as nossas estratégias de computação distribuída e paralela, de forma a otimizar esta etapa. Portanto, começou por se definir um intervalo de valores considerados 'normais' para os parâmetros em questão, tendo estes (SARIMA (p,d,q), (P,D,Q,M)) sido analisados individualmente no capítulo seguinte. Proposta de Solução. Tendo essa 'grid' de parâmetros definidos, como podíamos agora proceder à segmentação desta tarefa em sub-tarefas que possam ser distribuídas pelos nós disponíveis? A solução encontrada foi então, através dessa grid, distribuir a totalidade das possíveis combinações por esses nós da rede edge. No código seguinte é possível observar a criação dessa grelha de combinações, tal como a sua distribuição pelos nós via sc.parallelize():

Listagem 4.1: Criação de grelha e disitribuição dascombinações via RDD

```
sc = SparkContext(conf=conf)
#Grelha de combinacoes
p_values = range(0, 3)
q_values = range(0, 2)
P_values = range(0, 2)
Q_values = range(0, 2)
d = 1
D = 1
m = 24
param_grid = list(product(p_values, q_values, P_vaues, Q_values))
param_rdd = sc.parallelize(param_grid)
```

Listagem 4.2: Função de parametrização distribuída

```
def fit_auto_arima(params):
    from pmdarima import auto_arima
    import warnings
    warnings.filterwarnings("ignore")

    # Converte o Dataframe Spark 'df' de treino para lista local
    train_list = [float(row[var_a_prever]) for row in df.collect()]
    p, q, P, Q = params
    try:
        model = auto_arima(
            train_list,
            start_p=p, max_p=p,
            start_q=q, max_q=q,
            start_P=P, max_P=P,
            start_Q=Q, max_Q=Q,
            d=d, D=D,
            seasonal=True, m=m,
            stepwise=False,  #brute-force da combinacao
            suppress_warnings=True,
            error_action='ignore',
            trace=True,
            cache=False
```

```
        )
        return (model.aic(), (p, d, q), (P, D, Q, m))
    except Exception as e:
        return (float("inf"), (p, d, q), (P, D, Q, m))

# Executa a funcao paralelamente sobre o RDD e recolhe os resultados localmente
best = param_rdd.map(fit_auto_arima).reduce(lambda x, y: x if x[0] < y[0] else y)
# Escolhe o melhor
best = min(results, key=lambda x: x[0])
```

dado que o conjunto total de resultados corresponde ao número máximo de combinações (24, na matriz em questão), e considerando o facto de os resultados serem listas com combinações de parâmetros e respetivo AIC, sendo assim leves e não congestionando a memória local, seria também viável utilizar o método collect() em vez do reduce(), trazendo esses resultados para o driver e localmente verificar qual aquele que tem o menor AIC (a melhor combinação, portanto), ao contrário do reduce() que faz essas verificações de forma distribuída, ainda nos workers.

Relativamente ao nosso modelo de previsão, a opção recaiu sobre o modelo SARIMA, como já foi justificado anteriormente (capítulo 3). Apesar disso, continuam a existir algumas questões pendentes, sendo a mais relevante: ”de que forma iremos segmentar a execução do nosso modelo?”. A resposta acabou por recair em segmentar o número de variáveis (ou colunas) pedidas pelo utilizador, pelos nós disponíveis. Mas há certos cuidados a ter em conta: se o nosso modelo de previsão quer prever valores para um intervalo futuro breve, é essencial que este faça proveito de um conjunto de dados sequenciais a nível temporal, pois é um aspeto absolutamente fundamental na previsão linear de uma variável. Se um nó recebe apenas uma *slice* de dados respetivos ao intervalo ’1 a 15 de maio’, não faz sentido pedir uma previsão para, por exemplo, o intervalo de 5 a 8 de junho. O caminho escolhido foi, então, dividir o nosso dataset por colunas (ou seja, por variáveis), e encarregar cada nó de uma (ou mais) dessas variáveis. Suponhamos, o utilizador pede previsão a 1 dia de temperatura, humidade, pressão, e velocidade do vento (4 variáveis). Havendo, por exemplo, 4 nós (incluindo o master), cada um deles fica responsável por realizar a execução do modelo para a variável que lhe foi entregue. Para gerar e visualizar graficamente as previsões, foi utilizada a biblioteca mathplotlib.pyplot.

Listagem 4.3: Execução do modelo de previsão de forma distribuídalabel

```
import matplotlib.pyplot as plt
# Extrai os dados do dataframe para cada coluna e forma RDD
```

```
col_data = {
    col: [float(row[col]) for row in df.select(col).collect() if row[col] is
        not None]
    for col in target_cols
}
col_rdd = sc.parallelize(list(col_data.items()))

def col_sarima_forecast(col_tuple, best_params, steps):
    col, series = col_tuple
    p, d, q, P, D, Q, m =  best_params
    if steps ==24:  data_slice = 168        #1 semana de registos para treino
    elif steps == 72: data_slice = 168*2    #3
    elif steps == 24*5: data_slice = 168*4  #mes e meio
    try:
        treino = series[-168:]
        model = SARIMAX(
            treino,
            order=(p,d,q),
            seasonal_order=(P,D,Q,m),
            enforce_stationarity=False,
            enforce_invertibility=False
        ).fit(disp=False)
        forecast = model.forecast(steps=steps)
        return (col, series, forecast.tolist())
    except Exception as e:
        return (col, series, [])

#sarima_forecast_temp = sarima_forecast(train_list, steps=24)
results = col_rdd.map(col_sarima_forecast).collect()

#Criacao dos graficos
for col, series, forecast in results:
    plt.figure(figsize=(10, 4))
    plt.plot(range(len(series)), series, label="Historico")
    plt.plot(range(len(series), len(series)+len(forecast)), forecast, label="
        Previsao 24h", color='red')
    plt.xlabel("Hora")
    plt.ylabel(col)
```

```
39      plt.title(f"Previsao 24h para {col}")
40      plt.legend()
41      plt.grid(True)
42      plt.savefig(f"previsao_{col}.png", dpi=300, bbox_inches='tight')
43      plt.close()
```

Um aspeto interessante que se destaca na função col_sarim_forecast(), é o facto de o tamanho do conjunto de dados de treino se adaptar ao tamanho do intervalo de previsão, imprimindo no código alguma flexibilidade adicional e capacidade de adaptação ao contexto.

Figura 4.5: Previsão de temperatura horária no Rio de Janeiro

Relativamente à visualização das previsões obtidas, optou-se por gerar a apresentação de resultados em formato de gráfico. Apesar de não ser algo propriamente comum no contexto de previsões meteorológicas (como sabemos, se pensarmos numa previsão de temperatura para as próximas horas, tradicionalmente a visualização é feita no formato de tabela, como na fig. 4.5 por exemplo, ou na página de resultados da user interface que nós desenvolvemos e replicada na fig. 3.2(b)), nesta fase de desenvolvimento considerou-se que para uma compreensão mais rápida e eficaz das previsões promovidas e da relação desses resultados com os dados históricos analisados, a observação da mesma através de gráficos (fig. 4.6 para a velocidade do vento, ou fig. 3.7 para a temperatura, por exemplo), seria mais apropriado.

Estes testes preliminares, cujos resultados serão demonstrados e debatidos no capítulo 5, embora realizados sobre conjuntos de dados controlados e de pequena dimensão, mostraram-se essenciais para validar a comunicação entre nós e demonstrar a capacidade da plataforma em realizar previsões meteorológicas de forma distribuída. A sua estrutura modular permitirá facilmente a substituição ou

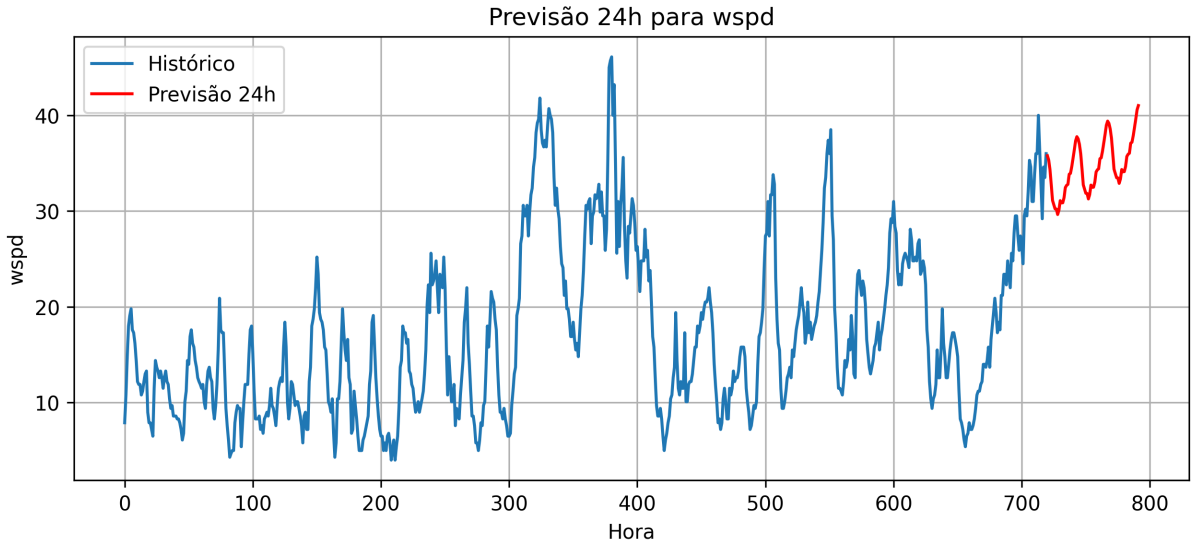


Figura 4.6: Previsão da velocidade do vento para os próximos 3 dias

evolução de componentes futuros — como novos modelos preditivos ou integração com fontes de dados externas.

5

Discussão de Resultados

Conteúdo

Neste capítulo, são apresentadas, primeiramente, as metodologias de avaliação do nosso sistema, de forma a definir como proceder à análise do nosso sistema, e examinados os resultados obtidos a partir da execução das diversas componentes da plataforma desenvolvida, particularmente as operações de parametrização, as funções de previsão meteorológica e o impacto da escalabilidade do cluster no desempenho global do sistema. As experiências realizadas tiveram como principal objetivo avaliar o comportamento da plataforma em diferentes configurações, tanto em ambiente de single-node como de multi-node, analisando a forma como o aumento do número de nós influencia a eficiência computacional, a latência e a capacidade de resposta da arquitetura. Adicionalmente, foram executadas funções de previsão com diferentes combinações de parâmetros e distribuições de carga, de forma a identificar eventuais bottlenecks no sistema, bem como as situações em que se verifica um ganho real de desempenho com o paralelismo e a colaboração entre nós.

O nó master, configurado com 6 GB de memória RAM, foi responsável pela coordenação do cluster

e execução do Spark driver. As máquinas virtuais constituintes do cluster, com sistemas operativos Linux (Ubuntu 22.04) e com 2 vCPU cada, estavam interligadas através de uma rede virtual em modo bridge, permitindo comunicação direta entre elas, simulando um ambiente de rede local (LAN).

Relativamente ao modelo de falhas do sistema, e considerando também a natureza dinâmica do nosso cluster, no qual os nós podem entrar ou sair de forma intermitente (*churn*), é importante perceber como o nosso sistema reage a tais situações. Ao nível do processamento de dados, em caso de falha de um nó e repentina indisponibilidade, o ambiente Spark, que oferece tolerância a falhas, redistribui automaticamente as tarefas que ficaram pendentes, para os restantes nós disponíveis, garantindo a continuidade da execução. Quanto ao armazanemanto, o HDFS assegura a recuperação e replicação dos blocos de dados em caso de falha de um DataNode e, assim, a perda de um nó não compromete a integridade dos dados armazenados.

5.1 Metodologia e Métricas de Avaliação

Nestas plataformas que temos vindo a falar, que possuem modelos preditivos na edge, torna-se também necessário aferir a qualidade desses mesmos modelos, como foi demonstrado na nossa proposta de solução, não só para garantir que o resultado mais fiável é entregue ao utilizador, como também para garantir que o modelo está otimizado e a funcionar corretamente. Para tal, basta dividir um determinado conjunto de dados, em subconjuntos de treino e de testes, comparando depois os resultados do modelo baseado no dataset de treino, com os resultados reais presentes no subconjunto para teste. Como avaliar agora esses resultados, para além de simples observações visuais?

5.1.1 Métricas de avaliação do modelo de previsão

Algumas das métricas de avaliação popularmente mais utilizadas para algoritmos de previsão são a *rooted mean square error* (RMSE), a *mean absolute error* (MAPE), ou o coeficiente de determinação (R^2) [39], descritas pelas fórmulas abaixo ilustradas.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(d_j - p_j)^2} \tag{5.1}$$

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n}|d_j - p_j| \tag{5.2}$$

$$R^2 = 1 - \frac{\sum_{j=1}^{n}(d_j - p_j)^2}{\sum_{j=1}^{n}(d_j - \bar{d})^2} \tag{5.3}$$

n é o número de amostras previstas (por exemplo, uma previsão de temperatura horária para o próximo dia, inclui 24 previsões, logo *n* = 24).

j o número da previsão atual.

d_j o resultado real/correto da previsão *j*.

p_j o valor previsto pelo sistema em causa para a previsão *j* (de um total de *n*).

O *mean square error*, Mean-Square Error (MSE), mede a média dos quadrados dos erros (diferença entre a previsão e o valor real). Quanto menor o valor de MSE, melhor o modelo. O Root Mean-Square Error (RMSE) é simplesmente a raiz quadrada do MSE e é mais facilmente interpretável no contexto aplicacional, pois é expressa nas mesmas unidades dos dados originais. Valores mais baixos de RMSE indicam um modelo mais preciso.

O *mean absolute error*, Mean Absolute Error (MAE), mede a média dos erros absolutos, ou seja, a média das distâncias entre a previsão e o valor real. Ele também é expresso nas unidades dos dados originais. Apesar de isso trazer facilidade de interpretação dos resultados, torna a métrica dependente da escala em questão, dificultando comparações entre variáveis. O Mean Absolute Percentage Error (MAPE), pelo contrário, expressa o erro em termos relativos, em formato percentual [39].

O *R-squared* (R^2), ou coeficiente de determinação, mede a proporção da variância de uma variável dependente, mas não é completamente fiável em modelos não-lineares e séries temporais. Varia entre 0 e 1 e quanto mais próximo o valor for de 1, mais precisa terá sido a previsão.

5.1.2 Avaliação da componente de computação em paralelo

As métricas mencionadas permitem-nos interpretar a eficiência dos nossos modelos de previsão. Resta agora avaliar outra componente importante do nosso sistema: a sua eficiência a distribuir e executar as tarefas necessárias pelos nós existentes, ou seja, a sua capacidade de aliviar e tornar a plataforma mais eficiente, através duma distribuição orquestrada pela camada fog de APs/LSs. Será necessário provar que a eficiência escala (tempos de execução reduzem) em função de fatores como o aumento de nós existentes na rede, por exemplo. Para isso, é necessário criar um cluster de número de nós variável, algo que pode ser alcançado com um *sofwtare* como o Docker. O Docker é uma plataforma *open-source* que utiliza *containers* para isolar aplicações e ambientes de execução, garantindo que estes se comportam de forma consistente e independente da infra-estrutura subjacente [21]. Cada nó será representado por um *container*, configurado para processar as tarefas atribuídas pela camada *fog*, que ficará responsável pela orquestração dos clusters. Assim, o Docker oferece as condições necessárias para simular clusters de nós independentes orquestrados por *master-nodes* (neste caso os elementos da camada fog), com características e condições individuais e personalizáveis, oferecendo a possiblidade de testar diferentes situações práticas, como variações do número de nós ativos, condições diferentes de rede e latência, ou na carga de processamento atribuída a cada nó em função desses mesmos parâmetros. Poderemos assim aferir a capacidade da nossa plataforma em lidar com diferentes contextos e dificuldades.

Os testes consistirão, inicialmente, na validação da funcionalidade básica da nossa camada de orquestração, ou seja, a sua capacidade em distribuir corretamente a carga computacional pelos nós existentes no cluster. Comparando o tempo de execução entre uma execução em *single-node*, exigindo a um utilizador que complete todas as tarefas individualmente, com uma execução feita a múltiplos nós (2, por exemplo, e assumindo que a capacidade computacional total do somatório desses 2 nós, seja superior à capacidade do *single-node* testado anteriormente), validaremos desde logo a capacidade da nossa plataforma em reduzir tempos de execução e latências ao distribuir as tarefas por diferentes nós que sejam capazes de contribuir, tirando partido de uma arquitetura de computação distribuída.

5.1.3 Testes de stress

Por fim, é interessante avaliar também o comportamento do sistema em função do escalamento dos dados necessários às tarefas que os nós terão de executar, de forma a perceber como o sistema lida com o aumento do volumes de dados. Mantendo reduzido o número de nós disponíveis (nó único ou apenas 2, por exemplo), aumentar sucessivamente a carga de processamento e aferir como o sistema reage até encontrar os seus limites. Primeiramente, realizar execução da previsão a 1 dia (intervalo mínimo) para apenas 1 variável, como temperatura, nesse número reduzido de nós. De seguida, fazer 2 execuções com duas e três variáveis selecionadas, aferindo como os tempos de execuções variam e se estes aumentam linearmente. Depois dessa verificação, realizar as mesmas execuções aumentando agora também os intervalos de previsão pretendidos (3 dias e 1 semana), aumentando assim os dados necessários para os cálculos. Assim testaremos situações de *stress* máximo para o sistema. Estes resultados possibilitarão também detetar possíveis *bottlenecks* e padrões da nossa plataforma que serão usados para refinar e aprimorá-la.

5.2 Parametrização do modelo de previsão

Nesta secção são apresentados os resultados obtidos relativamente à fase de parametrização do modelo de previsão utilizado na plataforma. O principal objetivo desta etapa foi testar diferentes combinações de parâmetros (aproveitando os parâmetros fixos definidos na seção 3.1), e execuções a single-node e também execuções distribuídas, de forma a identificar aqueles parâmetros que melhor equilibram a precisão da previsão e o **desempenho computacional** no contexto da arquitetura proposta, tal como verificar que os objetivos da plataforma em termos de eficiência são atingidos.

Foram realizadas execuções em diferentes configurações, com nós workers dual-core de 4 GB de memória, usando a variável temp (temperatura), variando não só os parâmetros internos do modelo, mas também o número de nós ativos no cluster, com o intuito de avaliar o impacto da distribuição das tarefas na performance global do sistema. Inicialmente, realizou-se a execução da função presente na

listagem 4.2 a single-node, utilizando um ficheiro .csv ('dados_meteo_small.csv' de 11347 bytes, ou seja, $\approx$ 11.1 KB, como é visível na fig. 5.1, resultante do comando 'hdfs dfs -ls') de 480 entradas. Após a falha,

Figura 5.1: Ficheiro em Hadoop Distributed File System (HDFS) utilizado pela função de parametrização

os nós são capazes de regressar ao estado 'Ativo' autonomamente, sem intervenção manual, apesar de logicamente não se obter qualquer resultado dessa execução e de os nós participantes ficarem indisponíveis durante alguns minutos (cerca de 20 minutos nos testes realizados, sem intervenção manual; caso se reinicie manualmente os nós, o tempo de indisponibilidade será reduzido).

As 3 primeiras execuções a 1 nó e com os parâmetros descritos de seguida, obtiveram os seguintes resultados:

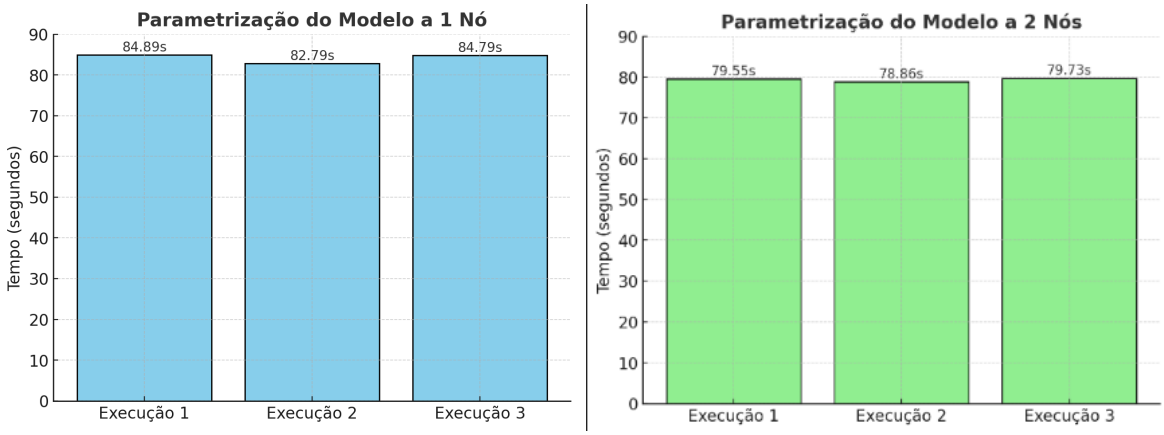
p = range (0, 3) ; q = range (0, 2) ; P = range (0, 2) ; Q = range (0, 2)

(1.4min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 84.89 segundos
(1.4min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 82.79 segundos
(1.4min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 84.79 segundos

De seguida, realizaram-se 3 novas execuções com a mesma grid de parâmetros, só que agora com 2 nós no cluster:
(1.3min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 79.55 segundos
(1.3min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 78.86 segundos
(1.3min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 79.73 segundos

Como podemos aferir, para além de identificar sempre o mesmo modelo como o mais indicado, o tempo de execução médio da função desceu de 84.16 segundos para 79.38s, concretizando uma redução de praticamente 5% (4.78 segundos), como se pode verificar nos gráficos da fig. 5.2. E se aumentarmos o 'scope' do grid, aumentando o range, por exemplo do parâmetro 'q' para (0, 3), como reagirá o nosso cluster?

Execução com apenas 1 nó a contribuir para a execução da função:
(3.3min) Melhor modelo encontrado:



(a) Execução centralizada **(b)** Execução partilhada

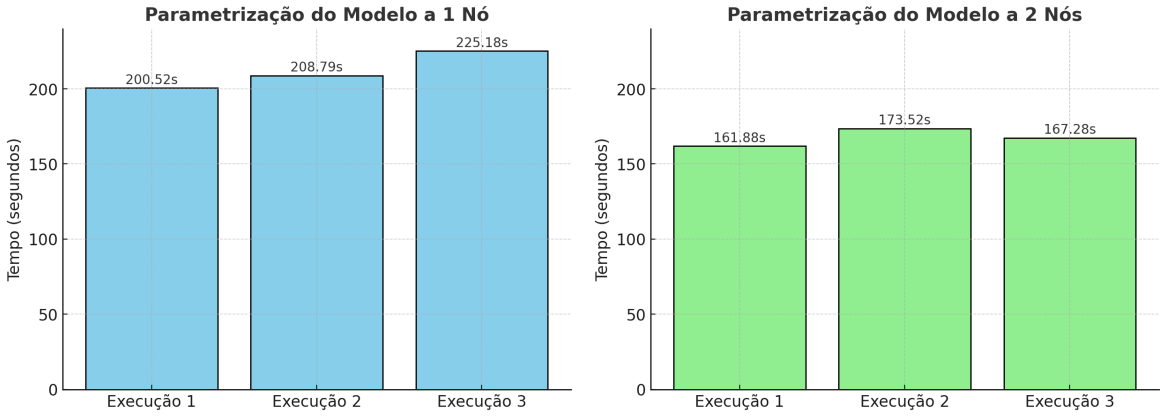
Figura 5.2: Função de parametrização do modelo com 24 combinações testadas

AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 200.52 segundos
(3.5min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 208.79 segundos
(3.7min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 225.18 segundos

Apesar da expansão da grid com novas combinações, a função de parametrização continua a identificar o mesmo conjunto de parâmetros como aquele que melhor ajusta o nosso modelo de previsão, reforçando a validade da seleção inicial de grid a utilizar. Vejamos agora como se comporta a plataforma realizando a mesma tarefa, só que com 2 nós a contribuir:

(2.7min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 161.88 segundos
(2.9min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 173.52 segundos
(2.8min) Melhor modelo encontrado:
AIC: 301.37 ARIMA: (2, 1, 1) Sazonal: (0, 1, 1, 24) Tempo de execução: 167.28 segundos

É fácil observar os ganhos consideráveis em termos de eficiência do sistema. Como era esperado, quanto maior a complexidade e quantidade de cálculos a realizar, mais reduções o nosso sistema é capaz de realizar relativamente aos tempos de execução, ao executar computações paralelas sobre os dados distribuídos (as combinações foram segmentadas via RDD, sendo que os dados de input são acedidos via HDFS).



(a) Execução centralizada

(b) Execução partilhada

Figura 5.3: Função de parametrização do modelo com 36 combinações testadas

Estes resultados revelam ainda que, embora o aumento do número de nós traga melhorias no desempenho, esse ganho não é linear nem garantido em todos os cenários. Em configurações com volumes de dados ou tarefas menos exigentes, o custo de coordenação entre os nós pode neutralizar os benefícios da distribuição. Por outro lado, em situações onde a complexidade dos parâmetros ou o tamanho da grid aumentam significativamente, a arquitetura distribuída demonstra clara vantagem, reduzindo o tempo total de execução de forma relevante. Esta observação reforça a importância de uma gestão dinâmica e criteriosa do offloading, que tenha em conta a natureza da tarefa, a capacidade dos nós disponíveis e o custo associado à paralelização.

5.3 Execução do modelo de previsão

Os testes relativos à componente da **eficiência da computação distribuída** do nosso modelo de previsão iniciaram-se com a execução do modelo nas suas condições máximas configuradas e com apenas 1 nó contribuidor, com o objetivo de realizar a previsão de 5 variáveis (temperatura, humidade, velocidade do vento, ponto de orvalho, e pressão atmosférica), para um intervalo de 5 dias e com um slice de treino de 4 semanas (24h * 7 dias * 4 semanas = 672 registos horários).

Tabela 5.1: Resultados em segundos dos testes de execução centralizada da função de parametrização

Nº de nós	Slice de treino	Intervalo de previsão	Tempo de execução
1	672	5 dias	24.07 s
1	672	5 dias	23.20 s
1	672	5 dias	22.09 s
1	672	5 dias	22.23 s
1	672	5 dias	22.11 s

Obteve-se uma média de tempo de execução de 22.74 segundos, como é possível verificar através dos resultados registados na tabela 5.1. O passo seguinte seria então verificar se existia uma redução destes tempos ao adicionar um nó worker/trabalhador ao nosso cluster.

Tabela 5.2: Resultados em segundos dos testes de execução distribuída da função de parametrização

Nº de nós	Slice de treino	Intervalo de previsão	Tempo de execução
2	672	5 dias	17.91 s
2	672	5 dias	19.26 s
2	672	5 dias	18.63 s
2	672	5 dias	17.95 s
2	672	5 dias	18.35 s

Podemos então observar que o tempo de execução médio neste tipo de condições (tabela 5.4), é de 18.42 segundos, inferior aos 22.74s registados anteriormente. Estes resultados são bastante positivos, uma redução superior a 4 segundos, valor que corresponde a cerca de 18.99%. Não restam dúvidas do aumento de eficiência do nosso ambiente, sabendo que também que noutras condições não existirão diferenças tão significativas (quase 20%), devido à redução da carga computacional.

Estando este teste concluído e obtidos os resultados expectáveis, decidiu-se reduzir progressivamente a complexidade das tarefas até possivelmente encontrar um ponto em que as execuções paralelas não se justificam. Mantiveram-se então as mesmas 5 variáveis 'alvo', reduzindo o intervalo de previsão para 3 dias (72 registos futuros, portanto).

Tabela 5.3: Resultados em segundos dos testes de execução centralizada da função de previsão

Nº de nós	Slice de treino	Intervalo de previsão	Tempo de execução
1	336	3 dias	12.63 s
1	336	3 dias	13.71 s
1	336	3 dias	13.74 s
1	336	3 dias	13.70 s
1	336	3 dias	13.77 s
1	504	3 dias	20.23 s
1	504	3 dias	20.50 s
1	504	3 dias	20.36 s
1	504	3 dias	20.44 s
1	504	3 dias	21.45 s
1	672	3 dias	24.83 s
1	672	3 dias	23.66 s
1	672	3 dias	23.03 s
1	672	3 dias	22.50 s
1	672	3 dias	23.84 s

Usando como input de treino a menor fatia de dados possível pensada para este intervalo de previsão, 336 registos horários que correspondem a 14 dias, e considerando em média o tempo de execução a 1 nó de 13.51 segundos, e de 15.67 segundos para execução a 2 nós, verifica-se que a 1 nó é produzida uma execução mais rápida. Considerando essa diferença de 2 segundos, e assumindo que 14 dias acaba por ser um conjunto bastante reduzido, testámos conjuntos de 503 (21 dias) e 672 (28 dias) registos, para verificar como reagia o nosso cluster.

Tabela 5.4: Resultados em segundos dos testes da execução distribuída da função de parametrização [REF]

Nº de nós	Slice de treino	Intervalo de previsão	Tempo de execução
2	336	3 dias	15.25 s
2	336	3 dias	15.37 s
2	336	3 dias	15.57 s
2	336	3 dias	15.93 s
2	336	3 dias	16.24 s
2	504	3 dias	22.02 s
2	504	3 dias	21.02 s
2	504	3 dias	20.62 s
2	504	3 dias	20.69 s
2	504	3 dias	23.49 s
2	672	3 dias	21.52 s
2	672	3 dias	20.25 s
2	672	3 dias	19.97 s
2	672	3 dias	20.99 s
2	672	3 dias	21.48 s

Para 504 registos de treino, o cluster acaba por apresentar resultados um pouco inconsistentes e, por vezes, piores ainda do que execuções com mais dados para treino (672), possivelmente devido a mecanismos internos do SARIMA que ocasionalmente poderá preferir arrays de maiores dimensões, não escalando os seus tempos totais linearmente em função do aumento dos dados de entrada. Utilizando a mesma slice de dados que é usada nas previsões a 5 dias, as melhorias são óbvias. Optaremos então por usar os 700 registos mais recentes para previsões de 3 dias.

Vamos reduzir o número de colunas (variáveis a calcular) para verificar o funcionamento do sistema a cargas menores, e a partir de quantas variáveis existem reduções de tempos de espera.

Tabela 5.5: Resultados de execução do modelo a 3 dias para 2 variáveis

Nº de workers	Tempo de execução
1	12.52 s
1	12.83 s
1	12.30 s
1	13.17 s
1	13.31 s
2	16.91 s
2	17.28 s
2	18.07 s
2	15.80 s
2	16.99 s

Não parece compensar paralelizar a execução para este esforço. Embora a paralelização traga ganhos de desempenho em muitos cenários, como se pode verificar pelos resultados da tabela 5.5, nem todas as tarefas, principalmente as mais pequenas e que exigem menos esforço, beneficiam da execução distribuída, e em alguns casos, esse custo adicional pode até prejudicar o desempenho. Ou seja, os encargos adicionais relacionados a essa distribuição de tarefas e recolha de resultados (a divisão dos dados, o envio das tarefas, a espera pelos workers e recolha dos resultados associados); este processo, sendo superior aos ganhos a nível da execução paralela das tarefas, acaba por custar mais tempos que aqueles poupados e ofuscar essas melhorias de eficiêncial, piorando a performance

do sistema.

Tabela 5.6: Resultados de execução do modelo a 3 dias para 3 variáveis

Nº de workers	Tempo de execução
1	15.92 s
1	15.63 s
1	15.42 s
1	15.34 s
1	15.80 s
2	18.95 s
2	19.46 s
2	19.95 s
2	19.96 s
2	20.38 s

Como se pode observar na tabela 5.6, ainda não compensa, pois so tempos de execução a single-node são inferiores a quando existe mais que 1 nó a contribuir para o processamento da tarefas. Procedemos então novamente ao aumento de mais 1 variável a prever, de forma a decidirmos a partir de que esforço compensará a computação paralela da tarefa em questão, cujos resultados estão visíveis na tabela 5.8.

Tabela 5.7: Resultados de execução do modelo a 3 dias para 4 variáveis

Nº de workers	Tempo de execução
1	18.52 s
1	17.98 s
1	17.67 s
1	18.72 s
1	17.95 s
2	18.28 s
2	15.90 s
2	18.63 s
2	16.50 s
2	16.28 s

Assim, utilizando 4 colunas do dataset como input (correspondendo no máximo a 2 colunas/variáveis por cada nó), já compensa a utilização da componente de processamento em paralelo, podendo-se observar uma redução de uma média de 18.17 segundos na execução a single-node para 17.12 segundos, ou seja, uma diferença de 1.05 segundos ($\approx 5.78\%$). Assim, a partir de pedidos que incluam intervalos de 3 dias e 5 dias, para 4 ou mais variáveis, valerá a pena fazer recurso das capacidades de processamento distribuído. Mas há que permitir ao utilizador fazer pedidos simples que não necessitem de tal, caso estes pretendam a maior velocidade de execução possível. Assim, os resultados obtidos acabam por confirmar que a execução distribuída deve ser aplicada de forma seletiva, com base em critérios como o volume de dados, a quantidade de variáveis previstas e os requisitos de tempo de resposta. A capacidade de adaptar dinamicamente o modo de execução consoante o contexto da tarefa permite que a plataforma funcione de forma eficiente tanto em dispositivos com recursos limitados, como em ambientes mais exigentes, como os que podemos encontrar em cenários de cidades inteligentes.

Agora, em caso de situações mais exigentes, ou **de stress** digamos, seja no volume de dados de input a analisar, na extensão das previsões a fazer, na instabilidade da rede, ou na limitação de capacidades computacionais, como reagirá a plataforma? Vamos então simular estas condições mais difíceis, começando por alterar o número de linhas utilizadas para treino pelo nosso modelo: foram pedidas previsões horárias de 6 variáveis (colunas) para os próximos 31 dias (24 * 31 steps portanto), com um input de 8664 linhas (registos horários dos últimos 12 meses para um determinado local). Foram também adicionados delays de 300ms a todas as interfaces de rede com o comando `'tc qdisc add dev ethx root netem delay 300ms'`. Com 1 nó, realizaram-se 3 tentativas de execução. Todas crasharam, por atingirem o limite máximo de failures permitidos aos workers. De seguida, adicionou-se 1 nó ao sistema, e tentaram-se testar os mesmos parâmetros para 2 datasets distintos: Londres (*dataset* 1), para os quais apresentou tempos de 156.33s e 205.20s; e Lisboa (*dataset* 2), com resultados obtidos em 329.33s e 196.10s (tabela 5.9). Chegámos portanto a uma situação em que o modelo só poderia ser executado com auxílio de dispositivos distintos, e através do processamento distribuido.

Tabela 5.8: Cenário onde é necessário recorrer a processamento distribuído

Nº de workers	Dataset	Tempo de execução
1	1	*crash*
1	2	*crash*
1	1	*crash*
1	2	*crash*
2	1	156.33 s
2	2	329.33 s
2	1	205.20 s
2	2	196.10 s

De seguida, de forma a sobrecarregar ainda mais a plataforma de forma a analisar o seu comportamento, adicionaram-se novas colunas [temp2, temp3, temp4, temp5, temp6], registos simulados de temperaturas horárias, ao dataset 1, de forma a aumentar o tamanho deste conjunto, e podermos pedir previsões para um elevado nº de variáveis [temp, dwpt, wdir, wspd, wpgt, pres, temp2, temp3, temp4, temp5, temp6], aumentando esse número de 6 variáveis pedidas para 11. Ao nível dessas previsões pedidas, mantivemos também a extensão da previsão pedida, pedindo previsões de registos horários para os próximos 31 dias (24 * 31 steps, portanto). Para executar estes testes, utilizaram-se inicialmente nós dual-core com limitação 1GB de memória disponibilizada para os *executors*.

Com 2 nós, tentou-se executar o script de previsão por 4 vezes, todas elas crashando durante a execução. Adicionou-se então ao ambiente, 1 novo nó idêntico aos restantes. Com esses 3 nós, foram pedidas novamente as previsões, as quais maioritariamente foram realizadas (1 das 5 execuções também crashou), apresentando os tempos observáveis na tabela 5.10. Como é possível verificar, os resultados, apesar de um pouco inconstantes, mostraram-se bastante positivos, demonstrando que o aumento de nós em contextos de tarefas muito exigentes é muito proveitoso para um sistema deste tipo.

Tabela 5.9: Testes de stress com executors a 1GB

Nº de workers	Tempo de execução
2	*crash*
2	*crash*
2	*crash*
2	*crash*
3	1701.57 s
3	926.34s s
3	1011.83s s
3	531.28s s

Tabela 5.10: Testes de stress com executors a 2GB

Nº de workers	Tempo de execução
2	1507.69 s
2	2381.09 s
2	1314.71 s
3	1874.30 s
3	1332.40 s
3	1629.71 s

Por fim, tentei aumentar ainda mais a complexidade das previsões pedidas, aumentando os steps exigidos para 24 * 31 * 3, ou seja, os 3 meses seguintes, aumentando também um pouco a capacidade dos nós, possibilitando agora a utilização de 2GB para RAM, duplicando assim o valor/capacidade anterior, de forma a dar possibilidade de o sistema a 2 nós também concluir as tarefas. Temos, portanto, uma média de 1612.14s para a execução a 3 nós, e uma média de 1734.50s para execuções a 2 nós. A diferença, apesar de não ser muito (redução de aproximadamente 7.05%), mostra um aumento de eficiência e da estabilidade dos resultados.

Estes testes acabam por mostrar que mesmo com cargas irrealistas para o nosso cenário, o aumento de nós aumenta a resiliência do sistema, pois torna-o mais resistente a falhas, acabando por conseguir concluir tarefas para as quais não se mostrava capaz com um número menor de nós (tabela 5.9). Aumentando a capacidade destes nós integrantes, passamos também a verificar essa capacidade para apenas 2 nós (tabela 5.10), apesar dos resultados serem menos eficientes, devido à menor capacidade de processamento total do sistema.

6

Conclusão

Conteúdo

Este trabalho visou explorar as principais dificuldades e soluções associadas à integração de tecnologias de computação distribuída em edge, enquadradas num contexto de tecnologias IoT ou potencialmente presentes em smart cities. A plataforma proposta, que se materializou num sistema de previsões meteorológicas de arquitetura fog em dispositivos em edge, acaba por ser um somatório de todas as conclusões depreendidas ao longo da investigação, tentando suprir os problemas habitualmente encontrados e aproveitando as tecnologias indicadas para lidar com este tipo de desafios.

A revisão de trabalhos relacionados demonstrou o potencial da conjugação de paradigmas de processamento local, com arquitetura fog, para lidar com computação de grandes volumes de dados ou de *streams* de dados em tempo real, descentralizando o processamento e aproximando as operações do utilizador e das fontes geradoras de dados.

Além disso, no que diz respeito especificamente à componente de previsões meteorológicas, na fase de decisão pretendeu-se simplificar os algoritmos de cálculo, de forma a oferecer maior flexibilidade e escalabilidade à plataforma, ainda para mais sabendo que um dos principais objetivos seria aproximá-los do utilizador, sendo por isso necessário considerar uma elevada heterogeneidade de dispositivos.

Apesar dessas considerações, era também importante garantir a qualidade e fiabilidade dessas mesmas previsões. Foi também delineada uma metodologia de testes para avaliação do funcionamento da mesma.

O desenvolvimento da proposta, descrito no capítulo 4, permitiu transformar os conceitos teóricos explorados nos capítulos anteriores numa implementação prática e funcional. Através da definição de uma arquitetura funcional clara, foi possível estruturar as camadas edge e fog de forma lógica e eficaz, definindo responsabilidades distintas para cada tipo de nó, desde a recolha e solicitação de dados até ao armazenamento, coordenação e processamento distribuído. A configuração do ambiente de execução, com recurso às ferramentas Apache Hadoop e Apache Spark, evidenciou a importância da integração entre sistemas de armazenamento distribuído e motores de processamento em memória, permitindo um equilíbrio entre robustez e desempenho. A utilização do HDFS como infraestrutura de armazenamento, combinada com o poder computacional do Spark, contribuiu para uma plataforma leve, mas capaz de lidar com dados em volume e velocidade. O código desenvolvido serviu de base para os testes experimentais, possibilitando simular a execução e parametrização de previsões meteorológicas em diferentes configurações.

A discussão de resultados demonstrou, através de experiências controladas, como a parametrização do modelo, que o número de nós ativos influencia diretamente o desempenho do sistema, como seria de esperar. Observou-se que a distribuição eficiente de tarefas, embora nem sempre compense para operações de baixa complexidade, oferece vantagens claras em contextos mais exigentes.

Este trabalho reforça, portanto, a importância e adaptabilidade de arquiteturas fog em redes edge que usufruam de capacidades de processamento distribuído, para lidar com contextos que exijam o tratamento de grandes ou constantes volumes de dados (como tecnologias IoT em massa que podemos observar no cenário considerado das redes em smart cities) ou contextos que exijam respostas em tempos críticos, oferecendo uma gestão de recursos eficiente e maximizando a utilização dos recursos disponíveis.

6.1 Trabalho futuro

Por fim, importa referir que os resultados e a arquitetura aqui desenvolvidos poderão ser adaptados ou estendidos para diferentes domínios de aplicação, especialmente em contextos onde a resposta em tempo real e a autonomia local são críticas. A plataforma poderá evoluir, por exemplo, para incorporar técnicas mais avançadas de previsão, suportar um número maior de variáveis ou integrar mecanismos de decisão baseados em eventos. Fica assim aberta a possibilidade de continuidade deste trabalho, seja através da sua aplicação em ambientes reais de edge computing [40, 41], seja pelo alargamento das suas capacidades dentro do ecossistema das smart cities e da IoT. Como perspetiva de trabalho

futuro, seria interessante expandir a plataforma desenvolvida para testes em ambientes reais, com dispositivos heterogéneos e ligações de rede variáveis, de forma a validar a sua robustez em cenários de produção. Adicionalmente, a integração de modelos de previsão mais avançados, o aumento de dados de ingestão necessários para volumes massivos, bem como a automatização do processo de decisão de offloading com base em inteligência adaptativa, poderiam reforçar ainda mais a capacidade da solução para lidar com ambientes urbanos dinâmicos, típicos de redes de cidades inteligentes. Considerar também a introdução de mecanismos de aprendizagem federada, que possibilitariam a realização de previsões cooperativas preservando a privacidade dos dados dos utilizadores, reduzindo a necessidade de transferência de informação sensível, com integração de dados selectiva [42]. A ligação direta a sensores reais ou a sistemas de recolha de dados urbanos reforçaria o realismo e aplicabilidade do sistema, abrindo caminho para o desenvolvimento de soluções de previsão e deteção de eventos em tempo real no contexto das smart cities. Como perspetiva adicional de evolução, poder-se-ia ainda considerar a integração de mecanismos de comunicação direta entre os dispositivos edge, eliminando assim a dependência de infraestrutura de rede fixa. Tecnologias como o Wi-Fi Direct e Wi-Fi Aware (*Neighbor Awareness Networking*), por exemplo, permitem a comunicação peer-to-peer e a descoberta de nós próximos. A utilização destas abordagens permitiria reforçar a resiliência e a disponibilidade da plataforma em cenários de falha de conectividade da rede.

Bibliografia

[1] S. Yi, Z. Hao, Z. Qin, and Q. Li, “Fog computing: Platform and applications,” 11 2015.

[2] S. P. Singh, A. Nayyar, R. Kumar, and A. Sharma, “Fog computing: from architecture to edge computing and big data processing,” in *The Journal of Supercomputing*, vol. 75, no. 4, April 2019, pp. 2070–2105. [Online]. Available: https://doi.org/10.1007/s11227-018-2701-2

[3] M. Chiang and T. Zhang, “Fog and iot: An overview of research opportunities,” *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[4] Deepak, M. K. Upadhyay, and M. Alam, “Edge computing: Architecture, application, opportunities, and challenges,” in *2023 3rd International Conference on Technological Advancements in Computational Sciences (ICTACS)*, 2023, pp. 695–702.

[5] S. Mohanty, “Everything you wanted to know about smart cities,” vol. 5, July 2016, pp. 60–70.

[6] R. Osorio, M. Peña-Cabrera, I. Lopez-Juarez, G. Lefranc, and R. Tovar-Medina, “Smart semaphore using image processing,” 10 2017, pp. 1–7.

[7] S. Bhatnagar, G. Nahar, V. Maurya, R. Mathur, B. Student, and P. Scholar, “Smart grid -for smart cities,” August 2020.

[8] A. Alnoman, “Edge computing services for smart cities: A review and case study,” in *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, 2021, pp. 1–6.

[9] O. Akyıldız, İbrahim Kök, F. Y. Okay, and S. Özdemir, “A p4-assisted task offloading scheme for fog networks: An intelligent transportation system scenario,” *Internet of Things*, vol. 22, p. 100695, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660523000185

[10] L. U. Khan, I. Yaqoob, N. Tran, S. Kazmi, T. Nguyen Dang, and C. S. Hong, “Edge-computing-enabled smart cities: A comprehensive survey,” September 2019.

[11] S. Tian, W. Yang, J. M. L. Grange, P. Wang, W. Huang, and Z. Ye, “Smart healthcare: making medical care more intelligent,” *Global Health Journal*, vol. 3, no. 3, pp. 62–65, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2414644719300508

[12] A. M. Rahmani, T. N. Gia, B. Negash, A. Anzanpour, I. Azimi, M. Jiang, and P. Liljeberg, "Exploiting smart e-health gateways at the edge of healthcare internet-of-things: A fog computing approach," *Future Generation Computer Systems*, vol. 78, pp. 641–658, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17302121

[13] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar, and J. H. Abawajy, "Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study," *IEEE Access*, vol. 5, pp. 9882–9910, 2017.

[14] S. Bhatnagar, G. Nahar, V. Maurya, R. Mathur, B. Student, and P. Scholar, "Smart grid -for smart cities," August 2020.

[15] H. Reis, "Exploring barcelona's urban innovations: Superblocks, sustainable mobility, and future gis possibilities," March 2024. [Online]. Available: https://www.geoweeknews.com/blogs/barcelona-superblocks-sustainable-mobility-gis-geospatial-technology

[16] T. Nam and T. A. Pardo, "Conceptualizing smart city with dimensions of technology, people, and institutions," in *Proceedings of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times*, ser. dg.o '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 282–291. [Online]. Available: https://doi.org/10.1145/2037556.2037602

[17] H. K. Apat, R. Nayak, and B. Sahoo, "A comprehensive review on internet of things application placement in fog computing environment," *Internet of Things*, vol. 23, p. 100866, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660523001890

[18] F. C. Andriulo, M. Fiore, M. Mongiello, E. Traversa, and V. Zizzo, "Edge computing and cloud computing for internet of things: A review," *Informatics*, vol. 11, no. 4, 2024. [Online]. Available: https://www.mdpi.com/2227-9709/11/4/71

[19] S. Bebortta, S. S. Tripathy, U. M. Modibbo, and I. Ali, "An optimal fog-cloud offloading framework for big data optimization in heterogeneous iot networks," *Decision Analytics Journal*, vol. 8, p. 100295, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772662223001352

[20] M. Ka and K. .P, "Distributed computing approaches for scalability and high performance," *International Journal of Engineering Science and Technology*, vol. 2, 07 2010.

[21] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A containerized big data streaming architecture for edge cloud computing on clustered single-board devices," 05 2019.

[22] X. Zhang, L. Qi, and Y. Yuan, "Editorial: Convergency of ai and cloud/edge computing for big data applications," *Mobile Networks and Applications*, vol. 27, 06 2022.

[23] F. Costa, J. N. de Oliveira e Silva, L. Veiga, and P. Ferreira, “Large-scale volunteer computing over the internet,” *J. Internet Serv. Appl.*, vol. 3, no. 3, pp. 329–346, 2012. [Online]. Available: https://doi.org/10.1007/s13174-012-0072-0

[24] F. Costa, L. Veiga, and P. Ferreira, “Internet-scale support for map-reduce processing,” *J. Internet Serv. Appl.*, vol. 4, no. 1, pp. 18:1–18:17, 2013. [Online]. Available: https://doi.org/10.1186/1869-0238-4-18

[25] T. Morais, “Survey on frameworks for distributed computing: Hadoop, spark and storm,” 01 2015.

[26] The Apache Software Foundation, “Apache hadoop yarn documentation,” September 2024. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[27] P. Narula, “Hadoop yarn architecture,” April 2023. [Online]. Available: https://www.geeksforgeeks.org/hadoop-yarn-architecture/?itm_source=auth&itm_medium=contributions&itm_campaign=articles

[28] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, p. 107–113, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[29] I. Hashem, N. Anuar, A. Gani, I. Yaqoob, F. Xia, and S. Khan, “Mapreduce: Review and open challenges,” *Scientometrics*, vol. 109, 04 2016.

[30] A. Ajina, J. Christiyan, D. Bhat, and K. Saxena, “Prediction of weather forecasting using artificial neural networks,” *Journal of Applied Research and Technology*, vol. 21, pp. 205–211, 04 2023.

[31] E. Shaikh, I. Mohiuddin, Y. Alufaisan, and I. Nahvi, “Apache spark: A big data processing engine,” 11 2019, pp. 1–6.

[32] KnowledgeHut upGrad, “Components of apache spark (ecosystem),” September. [Online]. Available: https://www.knowledgehut.com/tutorials/apache-spark-tutorial/apache-spark-components

[33] P. Chen, A. Niu, D. Liu, W. Jiang, and B. Ma, “Time series forecasting of temperatures using sarima: An example from nanjing,” *IOP Conference Series: Materials Science and Engineering*, vol. 394, p. 052024, 08 2018.

[34] R. Meenal, P. Angel, D. Pamela, and E. Rajasekaran, “Weather prediction using random forest machine learning model,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 22, p. 1208, 05 2021.

[35] O. A., Z. A., and A. L., “Comparative analysis of weather prediction using classification algorithm: Random forest classifier, decision tree classifier and extra tree classifier,” *African Journal of Mathematics and Statistics Studies*, vol. 7, pp. 162–171, 05 2024.

[36] A. Ajina, J. Christiyan, D. Bhat, and K. Saxena, “Prediction of weather forecasting using artificial neural networks,” *Journal of Applied Research and Technology*, vol. 21, pp. 205–211, 04 2023.

[37] P. Paialunga, “Weather forecasting with machine learning, using python,” April 2021. [Online]. Available: https://towardsdatascience.com/weather-forecasting-with-machine-learning-using-python-55e90c346647

[38] G. Li and Y. Wang, “Automatic arima modeling-based data aggregation scheme in wireless sensor networks,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2013, no. 1, p. 85, Mar 2013. [Online]. Available: https://doi.org/10.1186/1687-1499-2013-85

[39] J. Cifuentes, G. Marulanda, A. Bello, and J. Reneses, “Air temperature forecasting using machine learning techniques: A review,” *Energies*, vol. 13, no. 16, 2020. [Online]. Available: https://www.mdpi.com/1996-1073/13/16/4215

[40] A. Pires, J. Simão, and L. Veiga, “Distributed and decentralized orchestration of containers on edge clouds,” *J. Grid Comput.*, vol. 19, no. 3, p. 36, 2021. [Online]. Available: https://doi.org/10.1007/s10723-021-09575-x

[41] C. Gonçalves, J. Simão, and L. Veiga, “A function-as-a-service middleware for decentralized collaborative edge computing,” *Future Gener. Comput. Syst.*, vol. 175, p. 108069, 2026. [Online]. Available: https://doi.org/10.1016/j.future.2025.108069

[42] P. Kathiravelu, A. Sharma, H. Galhardas, P. V. Roy, and L. Veiga, “On-demand big data integration - A hybrid ETL approach for reproducible scientific research,” *Distributed Parallel Databases*, vol. 37, no. 2, pp. 273–295, 2019. [Online]. Available: https://doi.org/10.1007/s10619-018-7248-y

A

Code of Project

Configaração dos ficheiros .xml para o ambiente Hadoop:

Listagem A.1: core-site.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3      <property>
4          <name>hadoop.tmp.dir</name>
5          <value>/home/hadoop/tmpdata</value>
6      </property>
7      <property>
8          <name>fs.default.name</name>
9          <value>hdfs://192.168.218.140:9000</value>
10     </property>
11 </configuration>
```

Listagem A.2: hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>dfs.data.dir</name>
        <value>/home/hadoop/dfsdata/namenode</value>
    </property>
    <property>
        <name>dfs.data.dir</name>
        <value>/home/hadoop/dfsdata/datanode</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Listagem A.3: mapred-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Listagem A.4: yarn-site.xml

```
    <configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
```

```
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>192.168.218.140</value>
    </property>
    <property>
        <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
        <value>1</value>
    </property>
    <property>
        <name>yarn.nodemanager.env-whitelist</name>
        <value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CONF_DIR,
        CLASSPATH_PERPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
    </property>
</configuration>
```

Código do frontend em Python para possível User Interface da aplicação, com recurso à biblioteca open-source 'streamlit':

Listagem A.5: User Interface em Python, com bibliotecas 'streamlit'

```python
import streamlit as st
import pandas as pd
import random

# Inicializa o estado da p[U+FFFD]gina
if 'pagina' not in st.session_state:
    st.session_state.pagina = 'inicio'

def vai_para_formulario():
    st.session_state.pagina = 'formulario'

def voltar_para_inicio():
    st.session_state.pagina = 'inicio'

def ir_para_previsao():
    st.session_state.pagina = 'previsao'

```

```
18 # PAGINA INICIAL
19 if st.session_state.pagina == 'inicio':
20     st.markdown("""
21         <style>
22             body { background-color: #1e1e1e; }
23             #MainMenu, footer {visibility: hidden;}
24             div.stButton > button {
25                 background-color: #4CAF50;
26                 color: white;
27                 border: none;
28                 border-radius: 60px;
29                 padding: 30px 60px;
30                 font-size: 28px;
31                 display: block;
32                 margin: 40px auto;
33             }
34         </style>
35     """, unsafe_allow_html=True)
36
37     st.markdown("<h1 style='text-align: center; font-size: 48px;'>Weather @
        Phone</h1>", unsafe_allow_html=True)
38     if st.button("Start!"):
39         vai_para_formulario()
40
41     st.markdown("<div style='height: 100px;'></div>", unsafe_allow_html=True)
42     st.markdown("<p style='text-align: center; font-size: 18px; color: gray
        ;'>Location: Lisbon</p>", unsafe_allow_html=True)
43
44 # FORMULaRIO
45 elif st.session_state.pagina == 'formulario':
46     if st.button("U+FFFD Voltar"):
47         voltar_para_inicio()
48
49     st.markdown("<h3 style='text-align: center; font-size: 24px;'>Intervalo
        da previsao</h3>", unsafe_allow_html=True)
50     intervalo = st.radio(
51         "Selecione o intervalo desejado:",
52         options=["1 dia", "3 dias", "5 dias"],
```

```
53          horizontal=True,
54          key="intervalo_previsao"
55      )
56
57      st.markdown("<div style='height: 20px;'></div>", unsafe_allow_html=True)
58      st.markdown("<h3 style='text-align: center; font-size: 24px;'>Variaveis a
            prever</h3>", unsafe_allow_html=True)
59
60      # Checkboxes para variaveis
61      variaveis_selecionadas = []
62      if st.checkbox("Temperatura", key="temp"):
63          variaveis_selecionadas.append("Temperatura")
64      if st.checkbox("Humidade", key="hum"):
65          variaveis_selecionadas.append("Humidade")
66      if st.checkbox("Velocidade do Vento", key="vento"):
67          variaveis_selecionadas.append("Velocidade do Vento")
68      if st.checkbox("Press[U+FFFD] Atmosf[U+FFFD]rica", key="pressao"):
69          variaveis_selecionadas.append("Press[U+FFFD] Atmosf[U+FFFD]rica")
70      if st.checkbox("Ponto de orvalho", key="orvalho"):
71          variaveis_selecionadas.append("Ponto de orvalho")
72
73      st.markdown("<div style='height: 30px;'></div>", unsafe_allow_html=True)
74
75      # Botao Submeter
76      col1, col2, col3 = st.columns([1, 2, 1])
77      with col2:
78          if st.button("Submeter"):
79              st.session_state.variaveis_escolhidas = variaveis_selecionadas
80              ir_para_previsao()
81
82  # P[U+FFFD]GINA DE PREVISAO
83  elif st.session_state.pagina == 'previsao':
84      st.title("Previs[U+FFFD]o por variavel")
85
86      horas = ["16h", "17h", "18h", "19h", "20h"]
87
88      for var in st.session_state.variaveis_escolhidas:
89          if var == "Temperatura":
```

```
90              valores = [random.randint(10, 35) for _ in horas]
91              unidade =[U+FFFD]C
92              formato = "{:.0f}"
93          elif var == "Humidade":
94              valores = [random.randint(40, 100) for _ in horas]
95              unidade = "%"
96              formato = "{:.0f}"
97          elif var == "Velocidade do vento":
98              valores = [random.uniform(0, 50) for _ in horas]
99              unidade = "km/h"
100             formato = "{:.1f}"
101         elif var == "Pressao atmosferica":
102             valores = [random.uniform(990, 1030) for _ in horas]
103             unidade = "hPa"
104             formato = "{:.1f}"
105         elif var == "Ponto de orvalho":
106             valores = [random.uniform(2, 20) for _ in horas]
107             unidade =[U+FFFD]C
108             formato = "{:.1f}"
109         else:
110             valores = ["-"] * len(horas)
111             unidade = ""
112             formato = None
113
114         df = pd.DataFrame([valores], columns=horas, index=["Valor"])
115
116         st.subheader(f"{var} ({unidade})")
117
118         if formato:
119             st.table(df.style.format(formato))
120         else:
121             st.table(df)
122
123     # Botao "Nova Previsao"
124     st.markdown("<div style='height: 30px;'></div>", unsafe_allow_html=True)
125     col1, col2, col3 = st.columns([1, 2, 1])
126     with col2:
127         if st.button([U+FFFD][U+FFFD]va Previsao"):
```

```
128             st.session_state.pagina = 'formulario'
129
130     # Localizacao (estatica), futuramente adquirir posicao GPS do dispositivo
            para identifica-la
131     st.markdown("<div style='height: 50px;'></div>", unsafe_allow_html=True)
132     st.markdown("<p style='text-align: center; font-size: 18px; color: gray
            ;'>Location: Lisbon</p>", unsafe_allow_html=True)
```

Código Python das funções distribuídas (já replicado durante o texto):

Listagem A.6: Execução do modelo de previsão de forma distribuída

```
1  import matplotlib.pyplot as plt
2  # Extrai os dados do dataframe para cada coluna e forma RDD
3  col_data = {
4      col: [float(row[col]) for row in df.select(col).collect() if row[col] is
           not None]
5      for col in target_cols
6  }
7  col_rdd = sc.parallelize(list(col_data.items()))
8
9  def col_sarima_forecast(col_tuple, best_params, steps):
10     col, series = col_tuple
11     p, d, q, P, D, Q, m =  best_params
12     if steps ==24:  data_slice = 168        #1 semana de registos para treino
13     elif steps == 72: data_slice = 168*3    #3
14     elif steps == 24*5: data_slice = 168*4  #mes
15     try:
16         treino = series[-168:]
17         model = SARIMAX(
18             treino,
19             order=(p,d,q),
20             seasonal_order=(P,D,Q,m),
21             enforce_stationarity=False,
22             enforce_invertibility=False
23         ).fit(disp=False)
24         forecast = model.forecast(steps=steps)
25         return (col, series, forecast.tolist())
26     except Exception as e:
```

```
        return (col, series, [])

#sarima_forecast_temp = sarima_forecast(train_list, steps=24)
results = col_rdd.map(col_sarima_forecast).collect()

#Criacao dos graficos
for col, series, forecast in results:
    plt.figure(figsize=(10, 4))
    plt.plot(range(len(series)), series, label="Historico")
    plt.plot(range(len(series), len(series)+len(forecast)), forecast, label="
        Previsao 24h", color='red')
    plt.xlabel("Hora")
    plt.ylabel(col)
    plt.title(f"Previsao 24h para {col}")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"previsao_{col}.png", dpi=300, bbox_inches='tight')
    plt.close()
```

Listagem A.7: Criação de grelha e disitribuição dascombinações via RDD

```
sc = SparkContext(conf=conf)
#Grelha de combinacoes
p_values = range(0, 3)
q_values = range(0, 2)
P_values = range(0, 2)
Q_values = range(0, 2)
d = 1
D = 1
m = 24
param_grid = list(product(p_values, q_values, P_vaues, Q_values))
param_rdd = sc.parallelize(param_grid)
```

Listagem A.8: Função de parametrização distribuída

```
def fit_auto_arima(params):
    from pmdarima import auto_arima
```

```
    import warnings
    warnings.filterwarnings("ignore")

    # Converte o Dataframe Spark 'df' de treino para lista local
    train_list = [float(row[var_a_prever]) for row in df.collect()]
    p, q, P, Q = params
    try:
        model = auto_arima(
            train_list,
            start_p=p, max_p=p,
            start_q=q, max_q=q,
            start_P=P, max_P=P,
            start_Q=Q, max_Q=Q,
            d=d, D=D,
            seasonal=True, m=m,
            stepwise=False,  #brute-force da combinacao
            suppress_warnings=True,
            error_action='ignore',
            trace=True,
            cache=False
        )
        return (model.aic(), (p, d, q), (P, D, Q, m))
    except Exception as e:
        return (float("inf"), (p, d, q), (P, D, Q, m))

# Executa a funcao paralelamente sobre o RDD e recolhe os resultados
    localmente
results = param_rdd.map(fit_auto_arima).collect()
# Escolhe o melhor
best = min(results, key=lambda x: x[0])
```