UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

GABRIEL BRAUN RODENBUSCH

DESENVOLVIMENTO DE API PARA PREDIÇÃO DA COMPLEXIDADE DE TEMPO DE EXECUÇÃO DE CÓDIGOS POR MEIO DE AUTOML

Joinville
2023

GABRIEL BRAUN RODENBUSCH

DESENVOLVIMENTO DE API PARA PREDIÇÃO DA COMPLEXIDADE DE TEMPO DE EXECUÇÃO DE CÓDIGOS POR MEIO DE AUTOML

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador: Dr. Ricardo José Pfitscher

Joinville
2023

Dedico este trabalho à minha mãe Janaína e meus avós Luiz Roberto e Maria Izabel.

AGRADECIMENTOS

Agradeço mais do que tudo à minha família. A possibilidade de entrar, me manter e eventualmente concluir um curso de graduação é apenas um dos itens mais recentes na imensa lista de coisas, a começar pela própria vida, que só obtive graças a eles. Por isso, agradeço principalmente à minha mãe Janaína Fontoura Braun, meu avô Luiz Roberto Braun e minha avó Maria Izabel Fontoura Braun, só vocês sabem o esforço que tiveram para eu estar aqui. Além deles, agradeço a todos os familiares que contribuíram, cada um de sua forma, para o meu desenvolvimento. Obrigado.

Gostaria de agradecer ao Programa Institucional de Bolsas de Iniciação Científica (PIBIC) pelo investimento na educação brasileira e pelo incentivo recebido em minha pesquisa, na forma de uma bolsa de iniciação científica.

Agradeço ao meu orientador, o professor Dr. Ricardo José Pfitscher, pois sem ele eu não teria nem bolsa, nem iniciação científica e nem TCC. Além do papel excepcional como professor nas matérias que tive o prazer de cursar com ele, demonstrou o mesmo, se não mais comprometimento em me orientar e ensinar fora da sala de aula. Graças a ele tive também a oportunidade de participar da publicação de um artigo com base em nossa pesquisa e a honra de apresentá-lo em um congresso em uma das experiências mais cativantes e enriquecedoras da minha vida.

Agradeço também aos membros da banca, composta pelos professores Dr. Benjamin Grando Moreira e Dr. Pablo Andretta Jaskowiak, não somente por aceitarem o convite para participar da banca mas também pelo zelo na correção e todas as sugestões que enriquecem o trabalho.

Gostaria de agradecer a todos os amigos, dos mais diversos semestres e cursos, que tornaram essa experiência mais agradável nem que só por um dia ou por uma aula. Ficarão na memória todos que andaram comigo pelos corredores da UFSC, seja no Perini ou na Prudente de Moraes. Se tentasse citar todos que me ajudaram ao longo da jornada, ou acabariam primeiro as páginas para escrever ou cometeria a injustiça de esquecer alguém. Dito isso, fica o meu agradecimento especial aos membros da Toca, que compartilharam tantas matérias comigo e me presentearam com tantos momentos. Falando em compartilhar e presentear, não poderia deixar de falar especificamente do Geromel, o amigo com quem mais compartilhei matérias, trabalhos e até um apartamento, e da Lígia, que me presenteou com tabuleiros de xadrez, muitas panelas, o passador de slides que usei na apresentação deste trabalho e o mais importante: sua amizade.

Quero agradecer à minha namorada, Victoria Devi Völker Köhler, primeiro por me amar e segundo por aguentar todos esses anos de faculdade comigo. Eu tenho certeza que jamais chegaria tão longe sem sua companhia e a mera chance desse diploma seria

apenas uma memória sem sua presença. Obrigado por todas os Monsters divididos, comidas provadas, longas caminhadas de uma casa para a outra, por me ligar tentando me acordar quando eu não ouvia o despertador, por escanear meus trabalhos e provas no EAD quando meu celular não dava conta, por me ouvir dezenas de vezes apresentando este trabalho e, mais ainda, ler outras tantas o mesmo. Além dessas coisas e das milhares de outras com as quais você me ajudou ao longo da graduação, obrigado também pela oportunidade de te ver se desenvolver dentro dessa universidade e alçar vôos cada vez mais altos. Eu amo você.

Por fim, gostaria de agradecer à Universidade Federal de Santa Catarina por cada momento, principalmente os mais difíceis, que mais do que o futuro engenheiro, ajudaram o formar o homem. Além da pesquisa, ensino e extensão, quero agradecer à UFSC por ter posto em meu caminho a maioria das coisas pelas quais agradeci acima.

RESUMO

A análise da eficiência de um código é fundamental para a detecção de gargalos de desempenho e otimização do uso de recursos. O cálculo da complexidade do tempo de execução, referente ao período que um algoritmo leva para resolver determinado problema em função do tamanho da entrada, é um dos principais meios de medição desse parâmetro. Entretanto, de acordo com o Problema da Parada, cunhado por Martin Davis, é impossível elaborar um método geral para determinar se um programa irá parar ou executar indefinidamente. Portanto, também é impossível escrever um programa que estabeleça o tempo de execução de outro programa. Desse modo, é necessário trabalhar com aproximações para a classificação dessa complexidade. Técnicas de aprendizado de máquina vêm sendo empregadas para classificar a complexidade com base em características de código, como o número de estruturas condicionais e de loops aninhados. O banco de dados disponível para esse tipo de pesquisa é restrito, tanto em variedade quanto tamanho. Por isso, este trabalho expande o número de amostras atuais mesclando datasets de trabalhos relacionados com um dataset próprio, o Augustives. Esses bancos de dados são usados no treinamento de modelos com Autogluon, um método de aprendizado de máquina automatizado, que além de treinar uma variedade de técnicas, elabora também ensembles que as combinam em busca de maior acurácia. A partir dessa classificação, atingiu-se um máximo de 79,05% de acurácia para classe de complexidade e de 88,02% para eficiência. Por fim, o modelo que apresentou a melhor relação entre acurácia e tempo de resposta foi utilizado para a criação de uma API(Application Programming Interface) Rest, chamada RTCC, capaz receber um programa e retornar sua classe de complexidade e eficiência.

Palavras-chave: Complexidade de Tempo de Execução. Características de Código. Aprendizado de Máquina. API Rest.

ABSTRACT

Analyzing the efficiency of a code is fundamental for detecting performance bottlenecks and optimizing the use of resources. The calculation of the runtime complexity, regarding the period taken by an algorithm to solve a given problem as a function of the input size, is one of the main ways of measuring this parameter. However, according to Martin Davis' Halting Problem, it is impossible to develop a general method for determining whether a program will halt or run indefinitely. Therefore, it is also impossible to write a program that establishes the runtime of another program. Thus, it is necessary to work with approximations for the classification of this complexity. Machine learning techniques have been employed to classify the complexity based on code characteristics such as the number of conditional structures and nested loops. The available database for this type of research is limited, both in variety and size. Therefore, this work expands the number of current samples by merging datasets from related works with a proprietary dataset, named Augustives. These databases are used in models trained with Autogluon, an automated machine learning method that, in addition to training a variety of techniques, also constructs ensembles that combine them in search of higher accuracy. Through this classification, a maximum accuracy of 79.05% was achieved for the complexity class and 88.02% for efficiency. Finally, the model that showed the best trade-off between accuracy and response time was used to create a RESTful API (Application Programming Interface), called RTCC, capable of receiving a program and returning its complexity class and efficiency.

Keywords: Runtime Complexity. Code Features. Machine Learning. API Rest.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS

ANN Artificial Neural Networks (Redes Neurais Artificiais)

API Application Programming Interface (Interface de Programação de Aplicação)

AST Abstract Syntax Tree

AutoML Automated Machine Learning

CoRCod Code Runtime Complexity Dataset

DL Deep Learning

DRF Django REST Framework

DRY Don't Repeat Yourself

GPU Unidade de Processamento Gráfico

HTTP Hypertext Transfer Protocol

LGPD Lei Geral de Proteção de Dados

MB Megabyte

ML Machine Learning

OOF Out-of-fold

ORM Mapeador Objeto-Relacional

REST Representational State Transfer

RTCC Running Time Complexity Calculator

SMOTE Synthetic Minority Oversampling Technique

URI Uniform Resource Identifier

WWW World Wide Web

XGBoost eXtreme Gradient Boosting Trees

LISTA DE SÍMBOLOS

Ω	Letra grega maiúscula para omega
ω	Letra grega minúscula para omega
Θ	Letra grega Theta
θ	Letra grega theta
$\bar{x}$	Símbolo matemático referente à média amostral
s	Símbolo matemático referente ao desvio-padrão amostral

SUMÁRIO

1 INTRODUÇÃO

Na computação, a eficiência na resolução de um problema implica em velocidade e uso mínimo de recursos. Um dos principais parâmetros para a análise da eficiência de um código é a complexidade do tempo de execução, que, de acordo com Kleinberg e Tardos (2006), quantifica o tempo que um algoritmo leva para processar em função da entrada. Conforme demonstrou Martin Davis no problema da parada[1] (Lucas, 2021), é impossível escrever um programa que possa examinar outro e dizer, em qualquer caso, se o mesmo vai parar ou executar indefinitivamente (Strachey, 1965).

Recentemente, técnicas de aprendizado de máquina têm sido aplicadas para estimar a complexidade de tempo de execução. O primeiro trabalho nessa área, intitulado *Learning Based Methods for Code Runtime Complexity Prediction* (Sikka *et al.*, 2020), avaliou vários modelos de machine learning (ML) na classificação da complexidade de uma base de códigos em Java. Este trabalho, além de revisitar o estudo de Sikka *et al.*, dá continuidade a uma pesquisa cujos resultados foram publicados no artigo *Estimating Code Running Time Complexity with Machine Learning* (Pfitscher *et al.*, 2023), que compara novas técnicas de aprendizado de máquina e expande o banco de dados. Mais informações acerca da produção científica prévia podem ser encontradas no Apêndice A.

Sikka *et al.* trabalham com um banco de dados restrito, composto por 931 códigos em Java. O baixo número de amostras utilizado prejudica não somente o desempenho dos métodos de classificação, mas principalmente seu poder de generalização; além disso, o banco de dados não contempla algoritmos de ordem superior a $O(n^2)$. A baixa amostragem de classes ineficientes dificulta a separação binária de eficiência do classificador, que é mais relevante do que a separação entre várias classes eficientes. Pfitscher *et al.* expandem o banco de dados para um total de 1325 códigos a partir de raspagem de dados feita por web scraping do site Geek for Geeks[2], que contém códigos em Java acompanhados de sua complexidade.

É geralmente aceito que os dados e características determinam o limite máximo do aprendizado de máquina, e que os modelos e algoritmos só podem aproximar esse limite (He; Zhao; Chu, 2021), ainda assim, a busca pelo modelo ótimo é essencial para a otimização dos resultados. Sikka *et al.* (2020) definem o modelo de florestas aleatórias, o Random Forest, como o de melhor resultado dentre 8 modelos de classificação, enquanto Pfitscher *et al.* (2023) também o definem como o melhor em comparação a outros dois, o eXtreme Gradient Boosting Trees (XGBoost) e Artificial Neural Networks (ANN).

[1] Apesar da fama recair sobre Alan Turing, o teorema da parada foi primeiramente proposto por Stephen C. Kleene e posteriormente demonstrado por Martin Davis (Lucas, 2021).

[2] Disponível em: https://www.geeksforgeeks.org

Este trabalho adiciona novos códigos ao banco de dados de Pfitscher *et al.* (2023), além de introduzir a classificação das classes por meio de técnicas de aprendizado de máquina automatizado (AutoML). Isso visa facilitar a implementação de modelos e proporcionar maior velocidade do processo de treinamento e teste. Após a definição do método de classificação, o mesmo é aplicado na criação de uma Interface de Programação de Aplicativos (API) de predição, chamada RTCC (Running Time Complexity Calculator), que permite que um usuário, aplicativo ou site possa, dado um código de entrada, receber sua classe de complexidade.

1.1 OBJETIVO

Para facilitar a análise de eficiência de um algoritmo, propõe-se neste trabalho os objetivos a seguir.

1.1.1 Objetivo Geral

Desenvolver uma API RESTful capaz de estimar a classe de complexidade de um código-fonte em Java.

1.1.2 Objetivos Específicos

- Ampliação dos bancos de dados disponíveis atualmente para este tipo de pesquisa, com o intuito da consolidação de um banco mais extenso e balanceado.
- Desenvolvimento de um método de extração de características de código.
- Investigação de um método de classificação mais assertivo do que os utilizados na literatura, utilizando métodos ainda não testados para esse tipo de aplicação pela comunidade científica.
- Projeto e implementação de uma API capaz de fornecer serviços de classificação de complexidade de código e eficiência.
- Avaliação da API com base em requisitos definidos na metodologia.

1.2 ORGANIZAÇÃO DO TRABALHO

O trabalho está disposto em quatro capítulos e organizado da seguinte forma:

- Capítulo 2 - Fundamentação Teórica: trata de estabelecer uma base conceitual e uma comparação com a literatura frente a cinco temas; a complexidade de tempo de execução, o banco de dados utilizado, a extração de características do algoritmo, os métodos de classificação e a aplicação Rest.
- Capítulo 3 - Metodologia: descreve os procedimentos de coleta e tratamento dos dados e as implementações de aprendizado de máquina e da API.

- Capítulo 4 - Resultados: apresenta e discute os resultados obtidos ao longo da pesquisa.
- Capítulo 5 - Conclusão: destaca as principais descobertas e avanços científicos gerados pelo trabalho, além de debater estudos oportunos e possíveis melhorias para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo estabelece os conceitos necessários para o entendimento da complexidade de tempo de execução de um algoritmo. Primeiro, o que constitui um algoritmo e qual seu impacto no desenvolvimento de software. A seguir, a análise dos algoritmos e como sua eficiência é medida e representada. A partir desses conceitos básicos, trata-se então dos métodos de predição da complexidade do algoritmo. Para isso, é feita uma introdução do aprendizado de máquina e como as características de código podem ser aplicadas ao machine learning.

A abordagem utilizada é melhor detalhada ao expôr trabalhos anteriores na área. A literatura guia desta pesquisa é constituída pelos artigos *Learning based Methods for Code Runtime Complexity Prediction* (Sikka *et al.*, 2020), o primeiro a utilizar a análise de características para predição, e *Learning based Methods for Code Runtime Complexity Prediction* (Pfitscher *et al.*, 2023), o mais recente estudo nesse campo.

Com as fundações teóricas estabelecidas, é então abordado o trabalho em sí, o método de ML empregado e a API, que emprega todos esses conceitos em seu funcionamento e é melhor detalhada na metodologia.

2.1 ALGORITMOS

Esta seção aborda o que é e como analisar um algoritmo. De acordo com Cormen *et al.* (2012), trata-se de uma sequência de etapas que transformam uma entrada em saída. O algoritmo é considerado correto se, para toda entrada, parar com a saída correta. Essa definição é precisa pois ele pode não apenas apresentar uma resposta errada, como também nunca parar.

O desempenho de um sistema depende tanto de algoritmos eficientes quanto de um hardware capacitado. Contudo, mesmo diante dos significativos avanços tecnológicos ocorridos nos últimos anos, que integram arquiteturas computacionais avançadas a redes de alta velocidade, os algoritmos permanecem tão essenciais, se não mais, do que em períodos anteriores. Cormen *et al.* ainda exemplificam a relevância desses programas em cenários práticos:

> Até mesmo uma aplicação que não exija conteúdo algorítmico no nível da aplicação depende muito de algoritmos. A aplicação depende de hardware rápido? O projeto de hardware utilizou algoritmos. A aplicação depende de interfaces gráficas de usuário? O projeto de qualquer GUI depende de algoritmos. A aplicação depende de rede? O roteamento em redes depende muito de algoritmos. A aplicação foi escrita em uma linguagem diferente do código de máquina? Então, ela foi processada por um compilador, um

> interpretador ou um montador, e todos fazem uso extensivo de algoritmos. Os algoritmos estão no núcleo da maioria das tecnologias usadas em computadores contemporâneos. (Cormen *et al.*, 2012, p. 23).

Além desses casos, o aumento expressivo do uso de máquinas inteligentes em ambiente fabril tornou a otimização de seu desempenho uma questão essencial para a indústria, especialmente com a ascensão da indústria 4.0. Dentre às características desse movimento, é possível citar a Internet das Coisas, a análise de dados e o uso de robôs autônomos, todas dependentes de algoritmos (Pereira; Simonetto, 2018). Em aplicações como essas, onde o lucro é inerente à eficiência e produtividade, a busca por redução no tempo gasto em cada processo é uma busca direta pela redução de custos.

2.1.1 Análise de algoritmos

Ao analisar um algoritmo, busca-se prever os recursos necessários para sua execução. Esses recursos variam entre memória, largura de banda, hardware de computador e tempo de execução, sendo esse último o objeto desta pesquisa. Normalmente, o tempo gasto por um algoritmo cresce conforme a entrada aumenta. Por isso, descreve-se o tempo de execução de um programa em função do tamanho de sua entrada (Cormen *et al.*, 2012).

Um algoritmo é mais eficiente que outro se seu tempo de execução do pior caso apresentar uma ordem de crescimento mais baixa. Devido a fatores constantes e termos de ordem mais baixa, um algoritmo com maior ordem de crescimento pode demorar menos tempo para pequenas entradas do que um algoritmo de ordem menor. Porém, para entradas suficientemente grandes, o algoritmo de menor ordem será executado mais rapidamente no pior caso (Krane; Raymer, 2003).

Para se medir a complexidade de tempo de execução, é preciso analisar e computar o comportamento assintótico dos algoritmos, ou seja, como se comportam conforme a entrada aumenta, sobretudo nos casos sem limitação de crescimento. Para isso, são utilizadas notações que representam a função que rege cada programa. As três notações assintóticas principais são *Big O*, *Theta* (Θ) e *Ômega* (Ω). Além delas, existem as notações o e *omega* (ω), menos utilizadas (Krane; Raymer, 2003).

- a) Notação Θ:

$$f(n) = \Theta(g(n)) \approx a = b \tag{1}$$

  É utilizada para analisar a complexidade do caso médio de um algoritmo e limita assintoticamente uma função acima e abaixo (Krane; Raymer, 2003);
- b) Notação O:

$$f(n) = O(g(n)) \approx a \leq b \tag{2}$$

O tempo de execução do pior caso de um algoritmo. Estabelece um limite superior para o tempo de execução para qualquer entrada (Krane; Raymer, 2003);

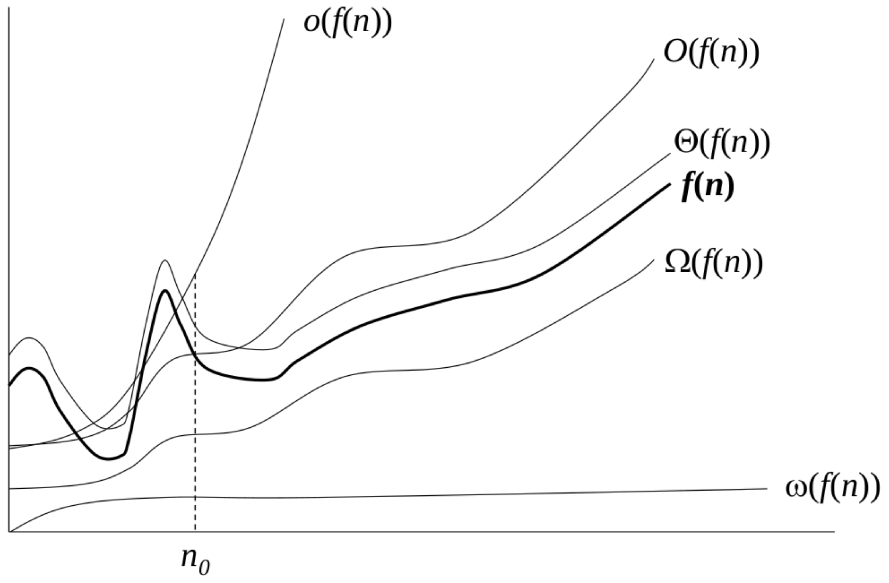
- c) Notação Ω:

$$f(n) = \Omega(g(n)) \approx a \geq b \tag{3}$$

Da mesma maneira que a notação O fornece um limite assintótico superior para uma função, a notação Ω dá um limite assintótico inferior (Krane; Raymer, 2003).

A Figura 1 apresenta um comparativo do funcionamento de cada notação em relação à uma função f(n). É importante ressaltar que neste trabalho a notação utilizada será sempre a *Big O*. Isso ocorre pois o interesse é na eficiência do código nos piores casos possíveis, pois a complexidade assintótica representa uma garantia de que esse é o tempo máximo que o código utilizará (Cormen *et al.*, 2012).

Figura 1 – Gráficos de cada notação



Fonte: (Krane; Raymer, 2003, p. 25)

Conforme esperado, a Figura 1 mostra a função f(n) por vezes crescendo mais rápido que a limitante O(g(n)). Entretanto, a partir de uma entrada de tamanho ($\mathbf{n}_0$), a função de O permanece superior para sempre. A velocidade de crescimento da função pode ser caracterizada por diferentes notações dentro da *Big O*. As mais relevantes para esse trabalho são analisadas ao longo da Subseção 2.1.2.

2.1.2 Classes de complexidade

Esta subseção aborda as classes de complexidade, que são divisões atribuídas a algoritmos de acordo com sua velocidade de crescimento no pior cenário. Classes consideradas eficientes apresentam um crescimento mais lento conforme sua entrada cresce em relação à classes ineficientes.

O Quadro 1 apresenta oito classes de complexidade guiadas pelo Big O. Entre essas, existem cinco classes consideradas pelo presente trabalho como eficientes e três ineficientes. Existem inúmeras classes de complexidade e várias formas de definí-las, as oito apresentadas a seguir foram as escolhidas por serem as mesmas utilizadas no artigo de Pfitscher *et al.*.

Quadro 1 – Notações de *Big O*

Notação	**Nome**	**Eficiência**
$O(1)$	Constante	Eficiente
$O(c^{\frac{1}{n}})$, tal que $n > 1$	Sublinear	Eficiente
$O(n)$	Linear	Eficiente
$O(\log n)$	Logarítmico	Eficiente
$O(n \log n)$	Linearítmico	Eficiente
$O(n^2)$	Quadrático	Ineficiente
$O(n^c)$, tal que $c \geq 3$	Polinomial	Ineficiente
$O(c^n)$, tal que $c > 1$	Exponencial	Ineficiente

Fonte: Adaptado de Chivers *et al.* (2015).

As cinco classes eficientes são:

- a) Constante: A saída dessa classe independe do tamanho da entrada. Sua constância a torna a mais eficiente das classes presentes (Woltmann, 2022);
- b) Sublinear: Equivalente à potência fracional, seu crescimento é proporcional à raiz enésima da entrada (Pfitscher *et al.*, 2023);
- c) Linear: Cresce linearmente ao tamanho da entrada. Seu tempo de execução é, no máximo, um fator constante vezes o tamanho da entrada. (Kleinberg; Tardos, 2006);
- d) Logarítmico: O algoritmo de busca binária é um exemplo bem conhecido. Esse padrão de tempo surge quando lida-se com algoritmos que realizam uma quantidade constante de trabalho para descartar uma fração constante da entrada (Kleinberg; Tardos, 2006);
- e) Linearítmico: Também chamado de quasilinear, é levemente mais ineficiente que a classe linear (Woltmann, 2022). É o tempo de execução de qualquer algoritmo que particiona a entrada em duas partes de tamanho igual, resolve cada parte de forma recursiva e depois combina as soluções em tempo linear

(Kleinberg; Tardos, 2006).

Por sua vez, as três classes ineficientes são:

- a) Quadrática: O tempo cresce conforme o quadrado do tamanho da entrada. O tempo quadrático pode surgir de um par de loops aninhados: Um algoritmo consiste em um loop com $O(n)$ iterações, e cada iteração do loop lança um loop interno que leva um tempo $O(n)$. A multiplicação desses dois fatores de n gera $O(n^2)$ (Kleinberg; Tardos, 2006);
- b) Polinomial: Kleinberg e Tardos (2006) descrevem $O(n^c)$ como um tempo de execução obtido para qualquer constante c quando procura-se em todos os subconjuntos de tamanho k. Embora a literatura assuma classes polinomiais como eficientes, este trabalho segue a definição do artigo de Pfitscher *et al.* (2023), que as estabelece como ineficientes visto que seu tempo de execução é muito superior à classe linear. Outra diferença é que Pfitscher *et al.* faz a separação da classe quadrática de outros casos polinomiais, de modo que o tempo polinomial é de $O(n^c)$, para $c > 2$. Para isso, parte-se do princípio que complexidades cúbicas são significativamente mais ineficientes que as quadráticas e, portanto, devem ser consideradas distintas.
- c) Exponencial: Por padrão, a classe exponencial se caracteriza por algoritmos cuja taxa de crescimento é exponencial em relação ao tamanho da entrada. Entretanto, assim como no artigo de Pfitscher *et al.* (2023), assume-se também como parte da classe exponencial os exemplos de complexidade fatorial. Esses dois tipos de limitação aparecem com frequência. Enquanto casos de $O(2^n)$ surgem naturalmente para algoritmos de busca que devem considerar todos os subconjuntos, espaços de busca de tamanho n! tendem a surgir ou como o número de maneiras de combinar n itens com outros n itens, ou como o número de maneiras de organizar n itens em ordem. A função fatorial cresce ainda mais rapidamente do que a exponencial, sendo assim, representa um limite ainda mais ineficiente (Kleinberg; Tardos, 2006).

Nessa seção foi definido como classificar algoritmos e as classes que esses podem ter dentro da notação Big O. Com a divisão em oito classes de complexidade de tempo de execução, estão também criados os rótulos a serem usados em uma classificação multiclasse. Além disso, sua divisão em classes eficientes e ineficientes pode ser replicada através da classificação binária.

2.2 APRENDIZADO DE MÁQUINA

Traçado o papel da complexidade para análise da eficiência, fica evidente sua relevância. Além disso, mostra-se necessário o desenvolvimento de meios para tornar o seu cálculo mais rápido e preciso. Enquanto a estimação de uma complexidade a

partir de elementos e estruturas dos códigos de forma manual é possível, esse método está sujeito tanto à falha humana quanto à lentidão. Por isso, esta seção analisa o uso de aprendizado de máquina para automatizar esse procedimento.

A máquina, além de lidar bem com uma grande quantidade de dados, ao contrário do ser humano, apresenta justamente seus resultados mais acurados conforme o banco de informações disponíveis aumenta (Zhou *et al.*, 2017). Esse comportamento, aliado à velocidade de suas análises, torna seu uso indicado para as mais variadas aplicações.

Conforme Sikka *et al.* (2020) citam em seu artigo, dois campos seriam favorecidos por esse método: ensino online e desenvolvimento de software. Cada vez mais, a educação é ofertada com componentes online. A possibilidade de realizar correções automáticas de atividades online envolvendo qualidade de código representaria uma melhoria significativa para o sistema educacional. O mesmo vale para desenvolvedores de software, que encontrariam nessa aplicação uma forma rápida de análise estática de seus códigos, o que proporciona um desenvolvimento mais eficiente no âmbito industrial.

2.2.1 MÉTODOS DE APRENDIZAGEM

A variedade de arquiteturas de aprendizagem é útil pois existem diversas possíveis aplicações de aprendizado de máquina. Por isso, esta subseção apresenta as arquiteturas mais convencionais e define qual técnica é empregada neste trabalho. Não há um método que seja ideal para todo caso, visto que cada característica matemática de um banco de dados pode ser melhor capturada por um diferente algoritmo. Além disso, fatores como o custo computacional, performance e a capacidade interpretativa da saída de um algoritmo de ML também impactam na escolha do modelo ideal.

Os métodos de aprendizagem de máquina mais utilizados são os de aprendizagem supervisionada. Nesses casos, a partir do treinamento de uma coleção de duplas (x, y), pode-se predizer um y* a partir de um valor de consulta x*. As entradas x podem variar de vetores a objetos complexos como documentos ou imagens. Da mesma forma, inúmeras abordagens para a saída y são possíveis, como classificação binária e multiclasse. Enquanto na binária y pode assumir dois valores, na multiclasse y pode assumir um dentre vários rótulos. A aprendizagem supervisionada prediz uma saída para cada entrada ou, em alguns casos, gera uma distribuição probabilística de y dado x, com base em um mapeamento aprendido f(x) (Jordan; Mitchell, 2015).

Uma das áreas de maior crescimento nos últimos anos no que tange aprendizado supervisionado é a das redes profundas. Os sistemas de aprendizagem profunda, do inglês deep learning (DL), utilizam algoritmos de otimização baseados em gradientes para ajustar os parâmetros de uma rede multicamadas de acordo com os erros de sua saída. Graças às Unidades de Processamento Gráfico (GPUs), esses

modelos lidam com bilhões de parâmetros e podem trabalhar com o imenso volume de imagens, vídeos e fala disponível atualmente (Audibert *et al.*, 2022).

Outra área de grande interesse é o estudo de algoritmos de DL capazes de identificar e desenvolver representações relevantes dos dados de entrada sem terem recebido alguma determinação de rotulação prévia. Esse método é conhecido como aprendizado não-supervisionado. Geralmente, essa forma analisa os dados não rotulados a partir de suposições sobre as propriedades estruturais dos dados, sejam elas algébricas, combinatórias ou probabilísticas (Murphy, 2012).

Essas técnicas de machine e deep learning, apesar de apresentarem resultados excelentes em diversas aplicações, ainda demandam muito da participação humana. He, Zhao e Chu (2021) ressaltam como mesmo modelos com milhões de parâmetros, que ocupam centenas de megabytes e requerem bilhões de operações de ponto flutuante ainda são projetados manualmente por especialistas em processos de tentativa e erro.

Frequentemente, quem necessita das análises e resultados obtidos pelo ML não possui conhecimento suficiente para aplicá-lo, ao passo que o contrário também ocorre. Karmaker *et al.* (2021) exemplificam essa divisão com especialistas de domínio e cientistas de dados. Segundo eles, os especialistas são fluentes no domínio onde o machine learning está sendo aplicado mas não são capazes de fazê-lo, enquanto os cientistas não conhecem o domínio da aplicação.

O aprendizado de máquina automatizado (AutoML) surge como uma alternativa capaz de tornar as aplicações mais acessíveis aos usuários finais. Por meio da automatização dos processos de modelagem do classificador, como a definição dos hiperparâmetros, muitas vezes feita por tentativa e erro, é possível melhorar o desempenho dos modelos, reduzir drasticamente o tempo de implementação e aumentar a reprodutibilidade e equidade de estudos científicos (Hutter; Kotthoff; Vanschoren, 2019). Isso, somado ao fato de que métodos mais tradicionais já foram aplicados em outros trabalhos relacionados a este projeto, torna o AutoML o método escolhido para esta aplicação.

2.3 TRABALHOS RELACIONADOS

Feita a introdução acerca do aprendizado de máquina, é preciso entender melhor como aplicá-lo à análise da complexidade de algoritmos. Para isso, esta seção primeiro apresenta uma discussão sobre o artigos *Learning Based Methods for Code Runtime Complexity Prediction* (Sikka *et al.*, 2020) e *Estimating Code Running Time Complexity with Machine Learning* (Pfitscher *et al.*, 2023). Após analisar a formação do banco de dados, extração de características de código e a análise comparativa entre diferentes modelos de classificação feita por ambos os artigos, o texto aborda o

repositório SACI[1], que tem um impacto significativo na metodologia do projeto.

2.3.1 Learning Based Methods for Code Runtime Complexity Prediction

O artigo *Learning Based Methods for Code Runtime Complexity Prediction* (2020) tinha o objetivo de desenvolver uma solução aproximada para programadores em busca de feedback em tempo real para a eficiência da complexidade de tempo de execução de seus códigos. Para isso, foi necessário elaborar o próprio banco de dados, ao qual foram aplicados modelos de classificação. Os resultados foram então comparados e analisados.

Devido à falta de um banco de dados de código aberto com códigos e suas respectivas complexidades de tempo de execução, Sikka *et al.* propuseram seu próprio dataset, chamado de *Code Runtime Complexity Dataset* (CoRCod), que foi extraído de uma plataforma de programação online. O CoRCoD[2] consiste de 931 arquivos de códigos referentes à cinco classes de complexidade, $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$ e $O(n^2)$, tidas como as principais pelos escritores do artigo. Esse banco de dados é considerado pelos mesmos o primeiro dataset público para o tema, e esse o primeiro trabalho que usa ML para predizer a complexidade (Sikka *et al.*, 2020).

Para a construção do dataset, foram coletados códigos-fonte em Java de diferentes problemas do site Codeforces[3], uma plataforma que regularmente hospeda competições de programação. A plataforma foi escolhida devido à alta variedade de estruturas de dados, algoritmos e complexidades de tempo de execução presentes. Para a extração dos dados, foi feito web scraping, técnica que extrai informações da World Wide Web (WWW) e as armazena análise futura. Devido à vastidão e variedade de dados presente na WWW, o web scraping é reconhecido como uma eficiente técnica para coleta de Big Data (Zhao, 2017).

Para garantir a veracidade dos dados obtidos, era necessário verificar a inexistência de erros de compilação e segmentação nos códigos. Para isso, foram filtrados apenas os códigos aprovados nos testes do próprio site. Frequentemente, foram coletados vários códigos para o mesmo problema, visto que eram soluções diferentes e, portanto, poderiam possuir complexidades diferentes. Os códigos foram então anotados manualmente por um grupo de cinco especialistas na área. Cada código era analisado por dois especialistas para redução de erro humano e os verificadores obtiveram complexidades iguais para os códigos em 100% dos casos.

As regras definidas durante o processo de anotação foram:

- a) Soluções com mais de uma variável de entrada, como n e m, e que possuem complexidade $O(n \cdot m)$, foram representadas como $O(n^2)$;

[1] Disponível em: https://github.com/Augustives/SACI
[2] Disponível em: https://github.com/midas-research/corcod-dataset
[3] Disponível em: https://codeforces.com

- b) Algoritmos de ordenação serão sempre de complexidade $O(n \log n)$;
- c) Inserções e recuperações em HashMap e HashSet serão consideradas como $O(1)$;
- d) Inserções e recuperações em TreeSet e Treemap serão consideradas como $O(\log n)$.

Os pesquisadores também removeram classes de complexidade com amostras insuficientes. Desse modo, o banco de dados definitivo é constituído de 932 códigos, obtidos a partir de cerca de 400 problemas, e que contêm 5 classes de complexidade. É possível verificar a distribuição dessas amostras na Tabela 1.

Tabela 1 – Distribuição de dados por classe

Classe de complexidade	**Número de amostras**
$O(n)$	385
$O(n^2)$	200
$O(n \log n)$	150
$O(1)$	143
$O(\log n)$	55

Fonte: (Sikka *et al.*, 2020).

Após a obtenção dos códigos, foi necessário extrair as características que impactam na complexidade de tempo de execução do algoritmo. Isso foi feito através de duas abordagens, a extração de características, também chamadas features, por meio de análise estática e o aprendizado de representações genéricas dos códigos a partir de code embeddings. No primeiro método, a extração ocorre a partir da identificação de construções de código chaves, como número de laços de repetição e presença de recursos. Essas features foram capturadas pelo Abstract Syntax Tree (AST) dos códigos, como é exemplificado na Figura 2. A AST se trata de uma árvore que representa as regras de sintaxe da linguagem utilizada, nesse caso, o Java (Sikka *et al.*, 2020).

Figura 2 – Extração da profundidade dos loops aninhados usando o ASTVisitor

```
class noOfNestedLoops extends ASTVisitor {
    int current = 0;
    int max_depth = 0;
    @Override
    bool visit(WhileStatement node) {
        current += 1;
        max_depth = max(current, max_depth);
        return true;
    }
    @Override
    void endVisit(WhileStatement node){
        current -= 1;
    }
}
```

Fonte: (Sikka *et al.*, 2020)

Seguindo o padrão de extração visto na Figura 2, Sikka *et al.* extraíram 28 características, dos quais as treze listadas no artigo estão descritos a seguir.

- a) Número de métodos;
- b) Número de switches;
- c) Frequência de condição-loop;
- d) Frequência de loops aninhados;
- e) Profundidade de loops aninhados;
- f) Número de variáveis;
- g) Número de instruções;
- h) Número de breaks;
- i) Número de loops;
- j) Frequência de loop-condição;
- k) Presença de recursão;
- l) Número de ifs;
- m) Número de jumps.

Com as features e banco de dados consolidados, os mesmo foram testados em sete modelos de classificação e no K-means. Os oito constam na Figura 3, assim como seus resultados.

Figura 3 – Algoritmos de classificação utilizados por Sikka *et al.*

Algorithm	Accuracy %	Precision %	Recall %	F1 score
K-means	50.76	52.34	50.76	0.52
Random forest	**71.84**	78.92	71.84	0.68
Naive Bayes	67.97	68.08	67.97	0.67
k-Nearest	65.21	68.09	65.21	0.64
Logistic Regression	69.06	69.23	69.06	0.68
Decision Tree	70.75	68.88	70.75	0.69
MLP Classifier	53.37	50.69	53.37	0.47
SVM	**60.83**	**67.62**	**67.00**	**0.65**

Fonte: (Sikka *et al.*, 2020)

A partir da Figura 3, fica visível a superioridade de desempenho do modelo de Random Forest. Esse modelo, chamado também de florestas aleatórias, apresenta tanto uma precisão de 78,92%, quase 10% a mais do que o segundo melhor modelo, quanto acurácia geral superior ao restante dos modelos, atingindo 71,84%. Com base nisso, Pfitscher *et al.* (2023) se basearam nesse modelo para suas implementações.

Com base nesses resultados, as principais contribuições desse artigo para este foram o banco de dados, que será chamado de Reference dataset, as features

utilizadas para classificação e as regras definidas para o processo de anotação.

2.3.2 Estimating Code Running Time Complexity with Machine Learning

Em relação ao trabalho anterior, Pfitscher *et al.*, cujo repositório do projeto está no Github[4], implementou duas mudanças principais: a inclusão de três novas classes de complexidade e a comparação de dois novos modelos de classificação com o modelo de melhor desempenho do artigo de Sikka *et al.* (2020), o Random Forest.

Enquanto Sikka *et al.* trabalhavam apenas com complexidades relativamente eficientes, o artigo *Estimating Code Running Time Complexity with Machine Learning* (Pfitscher *et al.*, 2023) expande o banco de dados com as classes polinomial e exponencial, que possuem desempenhos assintóticos bastante inferiores e são considerados ineficientes. Além das duas, também foi inclusa a classe sublinear, considerada eficiente.

Para incluir as novas classes, foi necessário realizar um novo scraping de dados. Para isso, foi desenvolvido um web crawler para captar códigos em Java presentes no site *GeeksforGeeks*[5], que contém um imenso acervo de programas e conteúdos relacionados à programação.

O crawler resultou em um dataset composto por 394 programas e 78 classes de complexidade distintas (Pfitscher *et al.*, 2023). A maioria dessas classes foi redistribuída manualmente para classes mais comuns, pois não seguiam a forma tradicional de classes assintóticas, o que resultou nas oito categorias presentes no Quadro 1: constante, linear, sublinear, linearítmica, logarítmica, quadrática, polinomial e exponencial.

Com os códigos-fonte captados, a extração de características foi feita por meio da biblioteca Javalang e resultou em um conjunto de oito features já presentes no artigo original de Sikka *et al.* e oito novas, listadas a seguir (Pfitscher *et al.*, 2023):

- a) Número de elses;
- b) Número de filas de prioridades instanciadas;
- c) Número de chamadas de busca binária;
- d) Número de chamadas para função de máximo e mínimo;
- e) Número de hash maps instanciados;
- f) Número de hash sets instanciados;
- g) Número de chamadas da função de ordenação;
- h) Número de chamadas recursivas para uma dada função.

O banco de dados extraído do *GeeksforGeeks* foi denominado Crawled dataset e somado ao Reference dataset, banco de dados formado por Sikka *et al.* (2020). A

[4] Disponível em: https://github.com/ricardopfitscher/RuTiCo
[5] Disponível em: https://www.geeksforgeeks.org/fundamentals-of-algorithms/

união desses dois bancos de dados formou o Merged dataset, que é o dataset principal do artigo.

As 16 features usadas por Pfitscher *et al.* foram selecionadas com base no pacote Boruta da linguagem R, para serem otimizadas de acordo com o banco de dados e a saída de interesse. Além disso, o banco de dados foi balanceado por meio de Synthetic Minority Oversampling Technique (SMOTE), responsável por melhorar o desempenho do algoritmo frente à dados muito desbalanceados, que possuem números de amostras desproporcionais para cada classe (Pfitscher *et al.*, 2023).

Utilizando o Merged dataset, foram comparados três modelos: Random Forest, Artificial Neural Networks (ANN) e eXtreme Gradient Boosting Trees (XGBoost). O Random Forest combina múltiplas árvores aleatórias de decisão e agrega suas previsões para gerar uma previsão final mais acurada e consistente (Biau; Scornet, 2016). O XGBoost é um sistema escalável de impulsionamento de árvores que combina algoritmos sensíveis à dispersão e esboço de quantis ponderados para aprendizado de árvores aproximado (Chen; Guestrin, 2016). Enquanto isso, as ANNs, também chamadas de redes neurais ou modelos conexionistas, consistem em camadas de neurônios artificiais que processam dados para realizar tarefas como classificação, previsão ou reconhecimento de padrões (Walczak, 2019).

Na Tabela 2 consta uma comparação entre o desempenho dos três modelos. Os três são comparados em relação à acurácia da classificação da eficiência e da classe de complexidade de tempo de execução dos códigos.

Tabela 2 – Comparação dos modelos de ML do artigo no Merged dataset

Objetivo	**Random Forest**	**XGBoost**	**ANN**
Eficiência	*90.17%*	90%	entre 85.5% e 87.5%
Classe de complexidade	*89.84%*	87.38%	entre 84.5% e 85.5%

Fonte: (Pfitscher *et al.*, 2023).

Como pode ser observado na Tabela 2, mesmo com mudanças no banco de dados e nas features, o Random Forest ainda se mostrou superior. Esse modelo é seguido de perto pelo XGBoost, que está mais próximo do primeiro lugar que do último, nesse caso, o ANN. Isso vai de encontro ao que dizem Sikka *et al.*, que citam como métodos de deep learning tendem a ser superados por métodos de machine learning para bancos de dados pequenos.

As maiores contribuições desse artigo foram as novas features usadas para classificação, a classificação binária da eficiência dos códigos, a extração de características por meio de Javalang, processo que é replicado neste trabalho, e os bancos de dados Crawled dataset (composto pelos códigos-fonte extraídos do *GeeksforGeeks*) e Merged dataset, formado pela junção do Crawled dataset elaborado

por Pfitscher *et al.* (2023) e do Reference dataset elaborado por Sikka *et al.* (2020).

2.3.3 SACI - Scraper of Algorithms and Complexitys

Esse repositório no Github, chamado SACI (Oliveira, 2015), também visa a extração de códigos e suas complexidades de tempo de execução por meio do scraping de dados. Assim como Pfitscher *et al.* (2023), esse projeto aplicou o web crawler no site *GeeksforGeeks*.

A partir de um total de 236 URLs do site, contendo 538 algoritmos ao todo, 511 algoritmos tiveram suas complexidades de tempo extraídas. Com base nos 511 extraídos com sucesso pelo projeto SACI, o presente Trabalho de Conclusão de Curso aplicou extração de características e desenvolveu um novo banco de dados, em um processo descrito no Capítulo 3. Sendo assim, a contribuição desse repositório para este trabalho foram os 511 que dão origem ao dataset Augustives, batizado com esse nome em homenagem ao dono do repositório SACI.

2.4 DISCUSSÃO

Os trabalhos abordados ao longo da Seção 2.3 apresentaram contribuições muito importantes para a área, principalmente no que tange banco de dados, metodologia de extração de características e classificação.

Embora este trabalho não pretenda, em um primeiro momento, introduzir um banco de dados com códigos inéditos para o tema, a extração das características dos códigos do SACI representa um avanço na pesquisa desse ramo pois o torna um banco de dados funcional. Isso, aliado à união dos bancos Augustives, Crawled e Reference, proporciona um treinamento mais assertivo de modelos para esse tipo de aplicação.

Sikka *et al.* (2020) afirmam que a limitação mais pertinente em seu artigo é o pequeno número de amostras no banco de dados. Além disso, é relevante notar as restrições oferecidas pelo número de classes de complexidade utilizados. $O(1)$, $O(n)$, $O(n \log n)$ e $O(\log n)$ são todos exemplos de classes eficientes. Classes ineficientes possuem uma complexidade de tempo de execução impraticável para grandes entradas e estão presentes nas mais diversas situações do cotidiano. Por isso, desenvolver um método de análise robusto para os dois tipos de classe de eficiência faz sentido e aumentaria a utilidade do projeto.

Aliando os datasets formados por cada projeto, espera-se obter um banco de dados capaz de performar a classificação de forma otimizada. Desse modo, a análise teórica feita ao longo desta seção foi fundamental para a consolidação do banco de dados do projeto. Além disso, é importante frisar que, ainda que por meio dos ajustes feitos por Pfitscher *et al.* (2023) na modelagem do Random Forest os resultados tenham evoluído, existem diversos métodos de classificação não testados.

Por isso, este trabalho aplicará o Autogluon, uma técnica de AutoML capaz de testar diversos modelos e escolher tanto o melhor modelo ou ensemble de modelos quanto os hiperparâmetros ideais para cada caso de forma autônoma.

2.5 AUTOGLUON

Esta seção trata do funcionamento do framework de aprendizado de máquina escolhido para ser utilizado na API. O AutoGluon é um framework de AutoML com ênfase em empilhamento automatizado de modelos. Para projetos que envolvem dados tabulares, como esse, é necessário utilizar seu submódulo tabular, o AutoGluon-Tabular.

Experimentos mostram que a combinação de várias camadas de muitos modelos oferece um uso melhor do tempo de treinamento alocado do que buscar apenas o melhor modelo. Um exemplo do potencial dessa técnica é que, em duas competições populares do Kaggle, otto-group-product-classification-challenge e bnp-paribas-cardif-claims-management, o AutoGluon superou 99% dos cientistas de dados participantes após apenas 4 horas de treinamento nos dados brutos (Bossan; Feigl; Kan, 2015, 2016).

2.5.1 Tratamento dos dados

O AutoGluon-Tabular automaticamente identifica se o problema é binário ou de múltiplas classes, assim como o tipo de cada coluna. Ele também lida com dados ausentes e balanceamento de features. Se não especificado, o AutoGluon divide automaticamente os dados em vários folds para treinamento e validação (AutoGluon, 2022).

Com base nesse tratamento de dados, os modelos são ajustados e os valores dos hiperparâmetros que produzem os melhores resultados são encontrados. O AutoGluon testa iterativamente valores para os hiperparâmetros para obter o melhor desempenho nos dados de validação, em um processo que pode ser paralelizado por várias threads ou máquinas. Em vez de apenas um único modelo, o AutoGluon treina vários modelos e os combina para garantir um desempenho preditivo superior (AutoGluon, 2022).

O AutoGluon depende de duas etapas sequenciais de processamento de dados: pré-processamento geral, que transforma as entradas para todos os modelos, e pré-processamento específico do modelo que é aplicado apenas a uma cópia dos dados usados para treinar um modelo específico (Erickson *et al.*, 2020).

O pré-processamento geral começa categorizando cada característica como numérica, categórica, texto ou data/hora. Colunas não categorizadas são descartadas dos dados e variáveis discretas ausentes são postas em uma categoria adicional

chamada "Unknown". Isso permite que o AutoGluon lide com categorias não vistas previamente no treinamento durante o teste (Erickson *et al.*, 2020).

2.5.2 Treinamento dos modelos

Para o treinamento, é usado um conjunto personalizado de modelos em uma ordem predefinida. Isso garante que modelos confiáveis e de alto desempenho, como Random Forest, sejam treinados antes de modelos mais custosos e menos confiáveis, como k-Nearest Neighbors. Além desses dois, o AutoGluon também considera redes neurais, LightGBM boosted trees, CatBoost boosted trees e Extremely Randomized Trees. No entanto, usuários podem adicionar modelos personalizados ao conjunto a ser treinado (Erickson *et al.*, 2020). Para esse trabalho, foram utilizados apenas esses seis modelos iniciais.

Conforme Sikka *et al.* (2020), métodos como as redes neurais de várias camadas utilizadas pelo AutoGluon tendem a desempenhar pior em datasets limitados. Isso, somado ao seu maior custo computacional se comparado à técnicas de ML, geralmente as torna descartáveis dos processos de treinamento de bancos de dados pequenos.

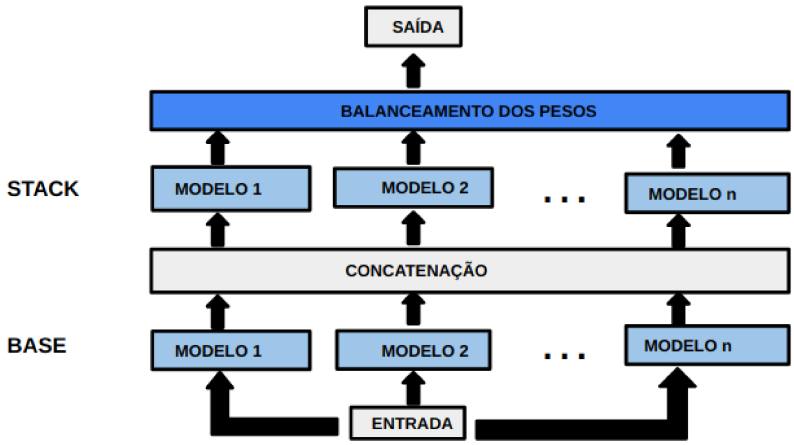
Por outro lado, Mendoza *et al.* (2016) demonstraram que redes neurais adequadamente ajustadas podem fornecer melhorias significativas na precisão quando adicionadas a um conjunto existente de outros tipos de modelos. Isso ocorre pois as fronteiras de decisão aprendidas por redes neurais diferem da geometria dos modelos baseados em árvores, proporcionando uma diversidade a modelos fruto de ensemble, o que justifica seu uso nesta aplicação.

2.5.3 Métodos de ensemble

Os frameworks de AutoML geralmente dependem de alguma forma de combinação de modelos, visto que ensembles são conhecidos por superar modelos individuais, muitas vezes reduzindo drasticamente a variância das previsões finais (Dietterich, 2000).

No Autogluon, primeiro uma coleção de modelos é treinada individualmente. Posteriormente, um multi-layer stacking é empregado usando as previsões agregadas dos modelos base, de modo que as previsões geradas por modelos stacker de camadas mais baixas sejam usadas como entradas para stackers de camada superior. Esse processo de iteração é aplicado pelo Autogluon conforme a Figura 4.

Figura 4 – Estratégia de stacking multi-layer do Autogluon, com duas camadas de stacking e n tipos de modelos de aprendizado base



Fonte: Adaptado de (Erickson *et al.*, 2020)

A saída mostrada na Figura 4 é a camada final de ensemble gerada pelo stacking do AutoGluon, que além de ser menos sujeita ao overfitting, possui variância reduzida em relação às camadas da base (Feurer *et al.*, 2015).

Por meio do k-fold bagging de todos os modelos em todas as camadas do stacking, a variância nas previsões finais é reduzida através da divisão aleatória dos dados em k partes distintas e, em seguida, do treinamento de k cópias de um modelo com uma parte diferente dos dados retida em cada cópia (Erickson *et al.*, 2020).

Durante o bagging, cada modelo produz previsões out-of-fold (OOF) das amostras não vistas durante o treinamento. Como cada exemplo de treinamento é OOF para uma das cópias do modelo "bagged", é possível obter previsões OOF de todos os modelos para cada exemplo de treinamento. No stacking, portanto, os modelos de camadas superiores treinam apenas com base nas previsões OOF das camadas inferiores, para evitar o overfitting (Erickson *et al.*, 2020).

Quando o AutoGluon recebe tempo de treinamento suficiente, o processo de k-fold bagging é repetido em n partições aleatórias diferentes dos dados de treinamento, calculando a média de todas as previsões OOF ao longo das bags repetidas. O número de repetições n é selecionado, se necessário, estimando quantas rodadas de bagging podem ser concluídas dentro do tempo de treinamento atribuído.

As previsões OOF que foram valores médios em várias k-fold bags exibem ainda menos variância e têm menos probabilidade de sofrer overfitting (Erickson *et al.*, 2020). Esse processo é útil em conjuntos de dados menores, onde o overfitting ocorre devido ao tamanho limitado dos dados OOF. Essa vantagem justifica o uso de AutoGluon neste trabalho.

O treinamento pode ser resumido à Figura 5. Cada camada de stacking recebe um orçamento de tempo total T(L). No Passo 7, o AutoGluon primeiro estima o tempo

de treinamento necessário e, se isso exceder o tempo restante para esta camada, pula para a próxima camada do stacking.

Figura 5 – Estratégia de treinamento: multi-layer stack ensembling + n-repeated k-fold bagging

Require: data (X, Y), family of models $\mathcal{M}$, # of layers L
Preprocess data to extract features
for $l = 1$ **to** L **do** {Stacking}
  for $i = 1$ **to** n **do** {n-repeated}
    Randomly split data into k chunks $\{X^j, Y^j\}_{j=1}^k$
    for $j = 1$ **to** k **do** {k-fold bagging}
      for each model type m in $\mathcal{M}$ **do**
        Train a type-m model on X^{-j}, Y^{-j}
        Make predictions $\hat{Y}_{m,i}^j$ on OOF data X^j
      end for
    end for
  end for
  Average OOF predictions $\hat{Y}_m = \{\frac{1}{n}\sum_i \hat{Y}_{m,i}^j\}_{j=1}^k$
  $X \leftarrow \text{concatenate}(X, \{\hat{Y}_m\}_{m\in\mathcal{M}})$
end for

Fonte: (Erickson *et al.*, 2020)

O algoritmo de treinamento descrito na Figura 5 apresenta um resumo do procedimento de geração de ensembles do AutoGluon. Esse procedimento é relevante pois, conforme será visto na metodologia, os resultados dos ensembles são os modelos que apresentam os melhores desempenhos de classificação.

Essa seção apresentou um resumo do funcionamento do AutoGluon, além de apresentar as inúmeras vantagens do mesmo em relação a modelos mais tradicionais de aprendizado de máquina. Resta introduzir os conceitos básicos de uma API, que embora sejam expostos inicialmente na Seção 2.6, são melhor compreendidos e expandidos durante a metodologia.

2.6 API RESTFUL

Após a introdução da teoria por trás da funções da API, é preciso compreender o que de fato é a API. Por se tratar de um tópico cuja relevância é principalmente nos aspectos práticos, esta seção apenas introduz alguns conceitos básicos por trás do desenvolvimento empregado no projeto, de modo que uma análise mais aprofundada da API é oferecida na Seção 3.5.

As Interfaces de Programação de Aplicações (APIs) permitem que desenvolvedores se aproveitem de funcionalidades já existentes para construir novas aplicações de maneira eficiente. Conforme exemplificam Robbes, Lungu e Janes (2019),

um aplicativo móvel pode depender de APIs para acessar sensores, componentes gráficos ou serviços na nuvem, enquanto os desenvolvedores podem se concentrar no que torna seu aplicativo verdadeiramente único.

Uma aplicação que possui um método padronizado para estabelecer a comunicação entre aplicativos ou sistemas diferentes, além de permitir o compartilhamento de dados, pode ser chamada de Web API. Existem diversas estruturas que podem ser utilizadas para definir o funcionamento da Web API. Este trabalho utilizada a arquitetura REST.

O Representational State Transfer (REST) é um estilo de arquitetura que define um conjunto de diretrizes para concepção de APIs para acessar e manipular recursos utilizando Hypertext Transfer Protocol (HTTP) , um protocolo de camada de aplicação (Golmohammadi; Zhang; Arcuri, 2022).

Numa API RESTful, qualquer pedaço de informação pode ser referido como um recurso. A fim de especificar a ação desejada para um determinado recurso, o HTTP define uma série de métodos de requisição designados por verbos HTTP (Golmohammadi; Zhang; Arcuri, 2022). Esses verbos, tais como GET e POST, permitem que recursos sejam criados, recuperados ou modificados (Gagliardi, 2021).

Em 2005, para acelerar o processo de desenvolvimento e automatizar tarefas na criação de APIs, foi criado o Django, um framework de aplicação web baseado em Python (Ghimire, 2020). Uma das principais vantagens do uso do Django é o seu Mapeador Objeto-Relacional (ORM), que institui modelos como a fonte definitiva de informações sobre dados da aplicação. Cada modelo contém campos e os comportamentos dos dados armazenados na API (Foundation; contributors, 2023).

Com a utilização do ORM, o Django busca seguir o princípio DRY (Don't Repeat Yourself), que enfatiza a reusabilidade e a máxima automatização dos processos. Além de seus funcionalidades intrínsecas, o Django possui também um framework especializado na construção de APIs REST, chamado Django REST Framework (DRF) (Gagliardi, 2021).

Dentre as funções do DRF, destacam-se os serializers, que serializam objetos Python, como os modelos, em formatos mais simples, como o JSON. Além disso, também é responsável por validar dados de entrada, gerenciar relações entre modelos, integrar views do DRF e personalizar dados de saída (Gagliardi, 2021).

A capacidade de criar páginas e formulários com pouco ou nenhum código destaca o Django e o Django Rest Framework. A velocidade de criação de formulários é vantajosa para um trabalho como este, que se propõe a lidar com uploads de arquivos. Por isso, a API RTCC foi desenvolvida utilizando arquitetura REST a partir do Django e do DRF (Gagliardi, 2021).

Com a definição da arquitetura da API, é concluída a introdução teórica, de modo que é possível avançar para as implicações práticas deste trabalho. Com isso,

encerra-se a fundamentação teórica e passa-se a investigar a metodologia do projeto.

3 METODOLOGIA

Neste capítulo é debatida a metodologia de pesquisa desenvolvida. Isso envolve os requisitos funcionais e não funcionais do projeto, bem como a criação e escolha das partes que formam a API, como os modelos de classificação, os bancos de dados e a extração de características. A partir desses, são elaborados diagramas de funcionamento da API, desde seu relacionamento com o usuário até como ela age internamente.

3.1 REQUISITOS DO PROJETO

De acordo com Sommerville e Sawyer (1997), requisitos são uma especificação do que deve ser implementado. Eles são descrições de como o sistema ou propriedade dele deve se comportar, podendo representar uma restrição no processo de desenvolvimento do sistema.

Por este projeto tratar de uma API em fase inicial, sem o intuito de disponibilizá-la imediatamente para o grande público, houve uma simplificação das necessidades da aplicação. Com base nisso, foram estabelecidos os requisitos funcionais e não funcionais descritos nas Subseções 3.1.1 e 3.1.2.

3.1.1 Requisitos funcionais

Os requisitos funcionais especificam os comportamentos que o produto exibirá em condições específicas, pois descrevem o que os desenvolvedores devem implementar para permitir que os usuários realizem suas tarefas (Wiegers; Beatty, 2013).

RF1 O sistema deve retornar ao usuário a estimativa da complexidade de tempo de execução baseada na notação Big O.

RF2 Um usuário deverá ser capaz de fazer upload de um arquivo na API.

RF3 A API deverá ser capaz de extrair as características do código em Java recebido.

RF4 O usuário cadastrado como superusuário deverá ser capaz de acessar a página de administração da API.

3.1.2 Requisitos não funcionais

Requisitos não funcionais geralmente tratam de atributos de qualidade e restrições que descrevem as características do produto em diversas dimensões que são importantes tanto para os usuários quanto para os desenvolvedores. Isso envolve

temas como desempenho, segurança, disponibilidade e portabilidade (Wiegers; Beatty, 2013).

Restrições de design e implementação impõem limitações às opções disponíveis para o desenvolvedor durante a construção do produto. Outros requisitos não funcionais abordam o ambiente em que o sistema opera, como plataforma, portabilidade, compatibilidade e restrições. Muitos produtos também são afetados por requisitos de conformidade, regulamentação e certificação (Wiegers; Beatty, 2013).

RNF1 A acurácia da classificação da eficiência do modelo utilizado na API deve ser superior a 80%.

RNF2 A acurácia da classificação da classe de complexidade do modelo utilizado na API deve ser superior a 70%.

RNF3 O tempo de resposta da API deve ser inferior a dois segundos.

RNF4 O tamanho do projeto após a compressão feita pelo Heroku deve ser inferior a 500 Megabytes.

RNF5 A API não deverá aceitar o upload de arquivos que não sejam .java.

RNF6 A API não deverá aceitar o upload de arquivos com tamanho maior que 1 Megabyte (MB).

RNF7 Após o envio do arquivo Java que não cumpra o **RNF5** ou o **RNF6**, a API deverá gerar automaticamente um alerta de erro legível e que especifique claramente o problema.

RNF8 Após o envio do arquivo Java que cumpre os requisitos **RNF5** e **RNF6**, a API deverá gerar automaticamente as strings referentes à classe de complexidade e à eficiência de forma legível.

RNF9 A coleta de informações e formação dos bancos de dados por parte da API deverá respeitar a Lei Geral de Proteção de Dados (LGPD).

RNF10 Um usuário não autorizado não deverá conseguir acessar a URI referente à administração da API.

Após a definição do que a API deve fazer e como deve fazê-lo, deve-se esclarecer como cada parte do sistema opera e o porquê disso. Sendo assim, primeiro trata-se da consolidação dos novos bancos de dados com quais o AutoGluon será treinado. Em seguida, ocorre a validação desse método de classificação para bancos de dados já utilizados na literatura, com o intuito de justificar o uso do AutoGluon em detrimento dos modelos usados em trabalhos relacionados.

Após a validação do AutoGluon, lida-se com o procedimento de extração de características dos códigos, processo vital para a própria construção do banco de dados criado na Seção 3.2. Por fim, o projeto da API é demonstrado para que se possa visualizar como esses elementos trabalham em conjunto.

3.2 BANCO DE DADOS

Esta seção aborda a consolidação dos bancos de dados do trabalho. Para isso, primeiro é desenvolvido um novo banco de dados a partir do repositório SACI, seguido da elaboração de dois novos bancos de dados com base na junção dos disponíveis até o momento.

Enquanto os bancos de dados utilizados por Pfitscher *et al.* (2023) já estavam prontos para uso, o projeto SACI apenas fornecia os códigos e sua complexidade. Para transformar essas informações em um banco de dados efetivo, existiram duas etapas. Primeiro, para um nível de confiança maior acerca dos dados, foi feita uma validação manual entre os códigos extraídos e suas respectivas características.

Dos 511 códigos obtidos inicialmente pelo web crawler do repositório SACI, apenas 431 haviam sido extraídos corretamente. Ou seja, sua complexidade extraída condizia com o valor referente ao código na página do *GeeksForGeeks*.

A segunda etapa foi, a partir desses códigos, extrair suas características para a criação das features do projeto. O processo de extração, feito pelo Javalang, é melhor abordado na Seção 3.4.

Dos 431 códigos analisados na API, 25 apresentaram erro. Isso representa 94,2% de sucesso na extração de características. Sendo assim, esse novo banco de dados, chamado de Augustives, contém 406 amostras. Na Tabela 3, observa-se a distribuição de classes obtida.

Tabela 3 – Distribuição de amostras por classe no Augustives dataset

Classe de complexidade	**Número de amostras**
Constante	31
Linear	182
Logarítmica	23
Linearítmica	55
Sublinear	2
Quadrática	78
Polinomial	19
Exponencial	16

Fonte: Elaborado pelo autor.

A partir das classes mostradas na Tabela 3, foi elaborado um novo banco de dados, chamado de Crawleds. Esse banco de dados é a união do Crawled original, resultado do web crawler de Pfitscher *et al.*, com o novo banco Augustives. Na Tabela 4 está disponível a distribuição das classes de complexidade desse novo dataset.

Tabela 4 – Distribuição de amostras por classe no Crawleds dataset

Classe de complexidade	Número de amostras
Constante	59
Linear	285
Logarítmica	72
Linearítmica	86
Sublinear	7
Quadrática	138
Polinomial	53
Exponencial	37

Fonte: Elaborado pelo autor.

A partir da distribuição de amostras da Tabela 4, é notável como mesmo com o aumento do banco de dados (737) e com a busca por classes mais ineficientes, as classes eficientes ainda representam 69,06% do total, de modo que o dataset continua relativamente desbalanceado.

Além disso, por serem dois crawlers aplicados ao mesmo site, existiam códigos presentes nos dois bancos de dados. Nesses casos, o código do Crawled de Pfitscher *et al.* (2023) foi excluído e o do Augustives mantido. Por fim, a presença da classe sublinear pode ser questionável visto suas poucas amostras, mas espera-se que isso proporcione uma comparação mais justa com o trabalho de Pfitscher *et al.*, que a utiliza.

Além desse banco de dados, também foi criado um que o une com o do artigo de Sikka *et al.* (2020). O dataset Mergeds, que pode ser visto na Tabela 5, representa a união do Crawleds com o Reference (2020). O Crawleds e o Mergeds são os dois bancos de dados analisados no contexto da API.

Tabela 5 – Distribuição de amostras por classe no Mergeds dataset

Classe de complexidade	Número de amostras
Constante	202
Linear	668
Logarítmica	127
Linearítmica	236
Sublinear	7
Quadrática	338
Polinomial	53
Exponencial	37

Fonte: Elaborado pelo autor.

A Tabela 5 apresenta um total de 1668 amostras, das quais 74,34% são representantes de classes eficientes. O aumento na eficiência da população em relação ao Crawleds é justificado pelo acréscimo do database de Sikka *et al.* (2020), que não

incluía duas das três classes ineficientes usadas atualmente.

Essa seção estabeleceu os dois novos bancos de dados a serem utilizados pela API. Enquanto o Crawleds une o Crawled dataset com o Augustives dataset e possui um total de 737 amostras, o Mergeds une o Crawleds com o Reference dataset e possui 1668, o que representa um aumento de 25,79% em relação ao número de amostras usado por Pfitscher *et al.* (2023) no Merged dataset, que era de 1326.

3.3 MODELO DE CLASSIFICAÇÃO

Esta seção aborda os passos necessários para a elaboração da melhor combinação possível para a otimização do classificador do AutoGluon. Embora o framework lide com a maior parte do trabalho de forma autônoma, a personalização de determinados parâmetros é encorajada pelos desenvolvedores do AutoGluon.

Ao utilizar a função fit() do AutoGluon-Tabular, responsável por todas as etapas de treinamento, quase nenhum ajuste manual é necessário. Entretanto, conforme o site oficial do framework (AutoGluon, 2023), para maximizar o desempenho do modelo final é necessário definir alguns parâmetros.

Optou-se por, na maioria das configurações, manter a configuração padrão do AutoGluon. Uma das implicações disso é o bagging não se repetir em vários k-folds, visto que, por padrão, o k é nulo. Além disso, a divisão de treinamento e teste também foi a padrão do AutoGluon, que é de 80 para treinamento e 20 para teste (AutoGluon, 2023).

O presente trabalho analisa a diferença nos resultados a partir do parâmetro de qualidade, também chamado de preset. Embora todos os presets utilizem os mesmos modelos de classificação apresentados na Subseção 2.5.2, eles se diferem principalmente pelo nível de sofisticação do ensemble de seus modelos. Os quatro presets de qualidade disponíveis são:

- a) `best_quality`:

  a.1) Melhor acurácia preditiva com pouca consideração ao tempo de inferência ou uso de memória.
- b) `high_quality`:

  b.1) Alta acurácia preditiva com rápida inferência;

  b.2) Com inferência de 10 a 200 vezes mais rápida e uso de memória de 10 a 200 vezes menor que o `[best_quality]`.
- c) `good_quality`:

  c.1) Boa acurácia preditiva com inferência muito rápida;

  c.2) Com inferência até 4 vezes mais rápida e uso de memória até 4 vezes menor que o `[high_quality]`.
- d) `medium_quality`:

d.1) Acurácia preditiva medíocre com tempos de inferência e de treinamento muito rápidos;

d.2) Com treinamento até 20 vezes mais rápido que o `[good_quality]`;

d.3) É o preset padrão do AutoGluon.

Além desses, outro preset relevante é o `[optimize_for_deployment]`. Utilizado para deletar modelos não usados e remover dados de treinamento, esse preset é capaz de otimizar o uso de memória em até 300%. Tendo em vista que o objetivo final é aplicar o AutoGluon em uma RESTful API e, portanto, realizar o deploy da aplicação, essa configuração faz sentido na versão final do projeto (AutoGluon, 2023).

É possível observar na Tabela 6 o desempenho dos modelos utilizando o Merged dataset, banco de dados desenvolvido por Pfitscher *et al.* no artigo *Estimating Code Running Time Complexity with Machine Learning* (2023) ao unir seu próprio banco de dados, o Crawled dataset, com o elaborado por Sikka *et al.*, chamado Reference dataset.

Tabela 6 – Comparação dos presets de treinamento para Merged dataset

Presets	**Acurácia para efficiency**	**Acurácia para complexity_class**
`best_quality`	88,38%	77,58%
`high_quality`	88,38%	77,58%
`good_quality`	88,38%	77,35%
`medium_quality`	86,79%	76,89%

Fonte: Elaborado pelo autor.

Conforme esperado, a Tabela 6 mostra o `[best_quality]` como parâmetro de melhor desempenho, obtendo 88,38% de acurácia para cálculo de eficiência e 77,58% para cálculo de classe de complexidade.

Estabelecendo o `[best_quality]` como modelo padrão, ao compará-lo na Tabela 7 com os resultados obtidos pelo Random Forest no artigo de Pfitscher *et al.* (2023), é possível notar um desempenho inferior. Entretanto, apesar da piora significativa para complexity_class, para efficiency a queda foi de menos de dois pontos percentuais.

Tabela 7 – Comparação do desempenho do AutoGluon e Random Forest para Merged dataset

Modelos	**Acurácia para efficiency**	**Acurácia para complexity_class**
`best_quality` AutoGluon	88,38%	77,58%
Random Forest (Pfitscher *et al.*, 2023)	90,17%	89,84%
Diferença de desempenho	1,79%	12,26%

Fonte: Elaborado pelo autor.

É necessário notar alguns fatores importantes ao analisar a Tabela 7. Primeiramente, enquanto para o Random Forest houve uma divisão de 70/30 para treinamento e teste, o AutoGluon treina com 100% dos dados. Além disso, é relevante também avaliar se esse menor desempenho é compensado por um menor overfitting.

A utilização de SMOTE para balanceamento dos dados antes da separação dos dados de treino e teste no artigo de Pfitscher *et al.* (2023) pode ocasionar vazamento de dados, o que aumentaria em muito o desempenho do modelo para teste com o dataset original, mas dificultaria a inferência a partir de novos dados. O AutoGluon-Tabular, além de não balancear os dados antes da separação, aplica outros métodos de minimizar o overfitting que o tornam mais confiável para generalização.

É possível observar melhor a generalização dos modelos na Tabela 8, onde compara-se o desempenho do AutoGluon com o Random Forest treinando o modelo com o Crawled dataset elaborado por Pfitscher *et al.* e testando com o Reference dataset elaborado por Sikka *et al.*.

Tabela 8 – Métricas para eficiência entre Random Forest e AutoGluon

Modelos	**Accuracy**	**Precision**	**Recall**	**F1-Score**
`best_quality` AutoGluon	84,425%	62,557%	68,5%	65,39%
Random Forest (Pfitscher *et al.*, 2023)	83,57%	83,91%	83,57%	83,72%
Diferença de desempenho	0,855%	21,353%	15,07%	18,33%

Fonte: Elaborado pelo autor.

A Tabela 8 mostra que, apesar das outras métricas apresentarem uma queda significativa, a acurácia para eficiência se manteve próxima. A perda de desempenho em precisão, recall e f1-score faz sentido levando em conta a disparidade de amostras do Reference dataset.

As amostras ineficientes representam menos de 30% do espaço amostral do banco de dados Reference e essas métricas tendem a baixar em ambientes desequilibrados. Por outro lado, a Tabela 9 mostra que a acurácia da predição da classe de complexidade foi 16% superior utilizando AutoGluon.

Tabela 9 – Acurácia para classe de complexidade entre Random Forest e AutoGluon

Modelos	**Acurácia**
`best_quality` AutoGluon	60,04%
Random Forest (Pfitscher *et al.*, 2023)	44,04%
Diferença de desempenho	16%

Fonte: Elaborado pelo autor.

Além disso, ao desempenho do modelo sofreu pouca alteração mesmo com

a troca de banco de dados. Em outro teste, ao inferir a classe utilizando o Crawled database tanto para treinamento quanto teste, o Random Forest apresentou 80% de acurácia e o AutoGluon apresentou 62,72%. Isso implica que o Random Forest atingiu 35,96% de queda ao tentar inferir a classe do Reference dataset.

Enquanto isso, a acurácia do AutoGluon passou para 60,04%. Ou seja, teve apenas 2,68% de queda no mesmo processo. Isso posto, a tendência é a generalização aumentar consideravelmente com a utilização do Merged dataset, visto que essa métrica melhora conforme o número de amostras cresce.

Agora que é possível aferir a validade dos modelos, os mesmos podem ser aplicados aos bancos de dados elaborados na Subseção 3.2.

3.4 EXTRAÇÃO DE CARACTERÍSTICAS DOS CÓDIGOS

Esta seção apresenta a extração de características utilizada pelo sistema. Conforme citado na Seção 3.2, foi necessário elaborar um método de extração para desenvolver um banco de dados a partir dos códigos captados no repositório SACI. Entretanto, a necessidade de extração é também fundamental para a implementação da API. Tendo em vista que o usuário da aplicação possui apenas um código-fonte em Java, faz parte das funcionalidades do sistema extrair essas informações e disponibilizá-las adequadamente para o modelo de classificação da API.

Novamente com o objetivo de facilitar uma comparação com trabalhos relacionados, além de simplificar a integração de bancos de dados, as características extraídas neste trabalho são as mesmas utilizadas por Pfitscher *et al.* (2023). As 16 características extraídas são:

- a) num_if: Número de ifs no código;
- b) num_else: Número de elses no código;
- c) num_switch: Número de switches no código;
- d) num_loof: Número de loops, incluindo for e while, no código;
- e) num_break: Número de breaks no código;
- f) num_priority: Número de filas de prioridade instanciadas no código;
- g) num_binSearch: Número de chamadas para busca binária no código;
- h) num_minMax: Número de chamadas para funções de mínimo e máximo no código;
- i) num_sort: Número de chamadas para funções sort no código;
- j) num_hash_map: Número de hash maps instanciados no código;
- k) num_hash_set: Número de hash sets instanciados no código;
- l) num_recursive: Número de chamadas de recursão no código;
- m) num_nested_loop: Profundidade máxima dos laços aninhados presentes no código;

- n) num_vari: Número de variáveis declaradas no código;
- o) num_method: Número de métodos declaradas no código;
- p) num_state: Número de statements presentes no código.

O processo de extração é feito em sua maioria através da Javalang, que é uma biblioteca do Python para análise de códigos-fonte em Java. Tipicamente, essa biblioteca é usada para analisar uma entrada de usuário por meio de uma representação interna. Ela proporciona um lexer e um parser focados no Java 8 (Thunes, 2023). O lexer da Javalang quebra uma string de entrada em uma sequência símbolos de vocabulário, chamados de tokens (análise léxica). O parser, por sua vez, constrói uma instância dessa representação interna aplicando uma estrutura gramatical a esses tokens (análise sintática) (Syme *et al.*, 2007).

É possível exemplificar o funcionamento da extração de informações pelo Javalang pela Listagem de código 3.1, que mostra a extração do número de ifs dos códigos-fonte.

```
1  import javalang
2
3  def count_ifs(content):
4      try:
5          tokens = javalang.tokenizer.tokenize(content)
6          count = 0
7          for token in tokens:
8              if isinstance(token, javalang.tokenizer.Keyword) and token.
       value == 'if':
9                  count += 1
10         return count
11     except javalang.parser.JavaSyntaxError as e:
12         print(f"Erro de sintaxe Java: {e}")
```

Listing 3.1 – Código-fonte if_counter.py

Na Listagem de código 3.1, o código-fonte Java fornecido como entrada, representado pela variável content, é tokenizado usando a função `javalang.tokenizer.tokenize()`. Os tokens podem ser palavras-chave, identificadores ou símbolos. O código itera sobre eles e, para cada token, verifica-se se ele é uma instância de `javalang.tokenizer.Keyword`, ou seja, uma palavra-chave, e se seu valor é "if". Se ambas as condições forem atendidas, a variável count é incrementada.

Para realizar a extração e armazenamento das informações, foi usada a própria API em seu ambiente de desenvolvimento. Após a extração da quantidade de cada característica, esses valores eram armazenados em um arquivo .csv, no mesmo formato encontrado nos bancos de dados vistos anteriormente. Um exemplo do formato do banco de dados Augustives está disponível no Apêndice B.

O desenvolvimento foi feito com o auxílio do Postman, uma plataforma de API

com ferramentas e processos integrados para a construção, gerenciamento, publicação e consumo de APIs (Inc, 2023). Por meio do Postman, foram feitos testes para diversos itens e requisitos da aplicação, conforme consta na Seção 3.5.

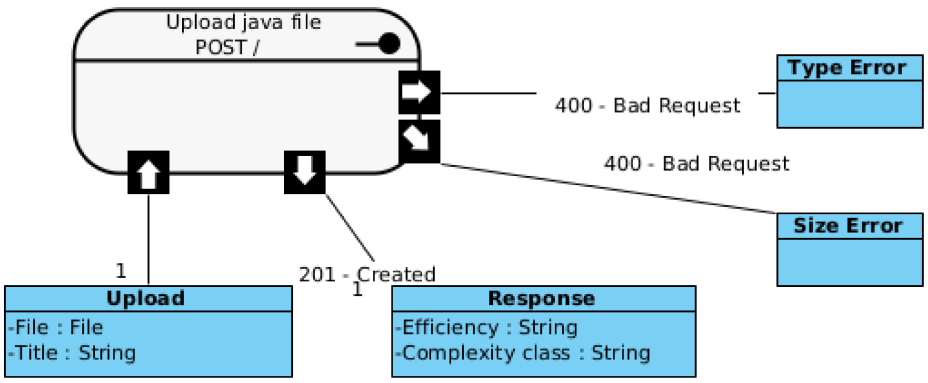
Nessa seção foi desenvolvido o processo de extração de características de forma bem-sucedida. Entretanto, conforme mencionado durante a formulação do banco de dados Augustives na Subseção 3.2, a extração de características dos códigos do repositório SACI falhou em 5,8% das vezes. Embora uma taxa de funcionamento de 94,2% seja consideravelmente alta, os 5,8% de falha confirmam que a API ainda não está pronta para um lançamento público, mesmo que todos os requisitos sejam cumpridos pelo resto dos resultados.

3.5 API REST

Esta seção busca, a partir dos pilares de desenvolvimento da API estabelecidos na Seção 2.6, esboçar o funcionamento da RTCC. Para a versão inicial da API, apenas o método HTTP POST é necessário. Por meio dele, o usuário cria um recurso Rest que envia dados ao servidor que serão processados de alguma forma (Golmohammadi; Zhang; Arcuri, 2022). No caso desta API, os dados enviados são um arquivo em Java e um título para esse arquivo.

Caso o usuário não disponibilize o título, a API atribui a ele o mesmo nome do arquivo .java. Após o processamento, obtém-se o resultado da classificação daquele código pelo modelo do AutoGluon e a API retorna sua classe de complexidade e sua eficiência. Um resumo dessa lógica pode ser encontrado no diagrama da Figura 25.

Figura 6 – Diagrama de recursos Rest



Fonte: Elaborado pelo autor

Conforme a Figura 25 mostra, caso o request do POST tenha sido feito corretamente, a response terá status 201, o que significa que o recurso foi criado

com sucesso. Caso tenha status 400, houve uma falha. Essa falha implica que ou o arquivo não era do tipo java, ou extrapolava o tamanho limite de 1 MB, utilizado para restringir ataques de negação de serviço.

A lógica que lida com a implementação dos procedimentos que lidam com os recursos REST é feita na `views.py` (Listagem de código 3.2), responsável por ou retornar um objeto HttpResponse contendo o conteúdo para a página requisitada, ou levantar uma exceção quando necessário (Foundation; contributors, 2023).

```
1
2  import json
3
4  from rest_framework import status
5  from rest_framework.response import Response
6  from rest_framework.views import APIView
7
8  from data.data_response import data_response
9  from data.models import Code
10 from data.serializer import CodeSerializer
11
12
13 class RunningTimeComplexityCalculatorAPIView(APIView):
14     queryset = Code.objects.all()
15     serializer_class = CodeSerializer
16     def get(self, request, format=None):
17         return Response(status=status.HTTP_405_METHOD_NOT_ALLOWED)
18
19     def post(self, request, format=None):
20
21         if 'file' not in request.FILES:
22             return Response({'error': 'File not sent'}, status=status.
    HTTP_400_BAD_REQUEST)
23
24         uploaded_file = request.FILES['file']
25
26         if not uploaded_file.name.endswith('.java'):
27             return Response({'error': 'The file must have a .java
    extension'}, status=status.HTTP_400_BAD_REQUEST)
28
29         max_size = 1024 * 1024  # Tamanho maximo permitido em bytes (1
    MB)
30         if uploaded_file.size > max_size:
31             return Response({'error': 'The file is too big. The maximum
    size allowed is 1MB.'}, status=status.HTTP_400_BAD_REQUEST)
32
33         content = uploaded_file.read().decode('utf-8')
34
```

```
35          response_data = data_response(content, uploaded_file)
36          result_json = json.loads(response_data)  # Converte a string
    JSON de volta para um dicionario Python
37
38          efficiency = result_json['Efficiency']
39          complexity_class = result_json['Complexity class']
40
41          response_data = {
42              'Efficiency': efficiency,
43              'Complexity class': complexity_class
44          }
45
46          return Response(response_data, status=status.HTTP_201_CREATED)
```

Listing 3.2 – Código-fonte views.py

Com relação ao servidor onde os recursos são criados, dois servidores foram usados, o próprio computador como um hospedeiro local e o Heroku[1], uma plataforma como serviço que disponibiliza um ambiente de computação em nuvem com suporte a várias linguagens e permite ao usuário criar e submeter suas aplicações (Solórzano; Charão, 2017).

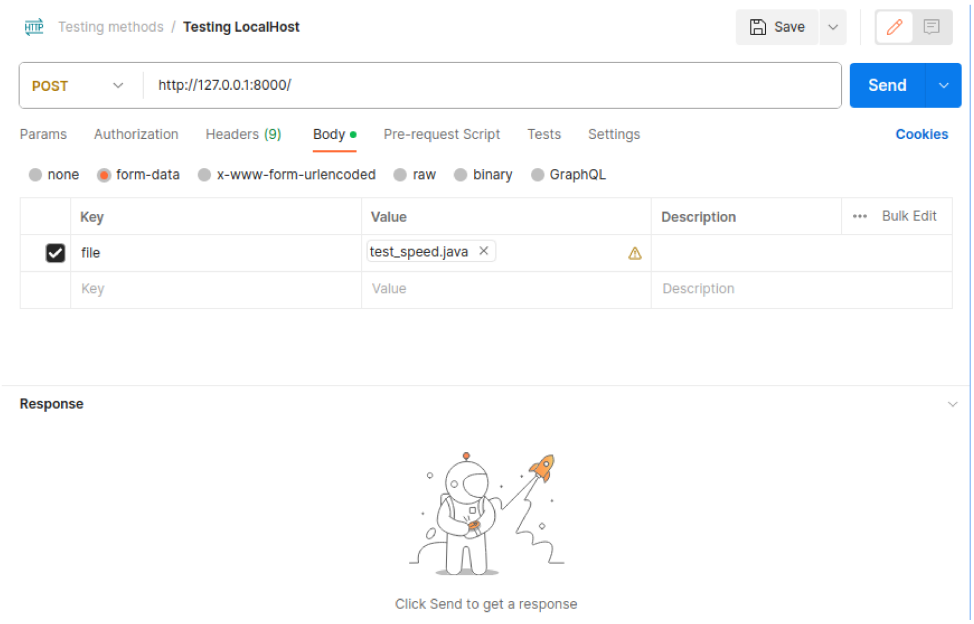
O Heroku foi escolhido como servidor do projeto por sua vasta popularidade para hospedagem de aplicações, o que tende a facilitar o processo de deployment, por se tratar de um serviço bastante documentado. Uma das limitações do projeto é que o Heroku possui, em sua versão base, um limite de ocupação de 500 MB para o aplicativo. Isso dificulta a utilização de bibliotecas mais robustas, como as usadas pelo preset `[best_quality]`. Por isso, ao longo do desenvolvimento foram testados métodos para redução do espaço de memória usado e tempo de resposta do RTCC visando o deploy da aplicação.

Para facilitar o desenvolvimento da API e os testes da mesma, utilizou-se o Postman. Na Figura 7 consta sua interface para testes no LocalHost, ambiente gerado por funções do próprio Django para o desenvolvimento do projeto.

Nos testes de envio para o Heroku, a única diferença é a alteração do endereço de envio de `http://127.0.0.1:8000/` para `https://rtcc-40c886516567.herokuapp.com`, domínio criado pelo Heroku para hospedar a aplicação. Após o POST, caso esse seja bem-sucedido, o response do sistema será semelhante ao da Figura 8.

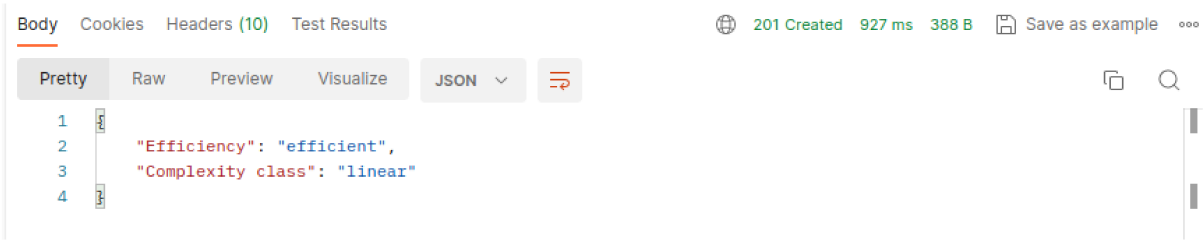
[1] Disponível em: https://www.dashboard.heroku.com

Figura 7 – Interface do Postman para envio ao LocalHost



Fonte: Elaborado pelo autor

Figura 8 – Interface do Postman para response do LocalHost

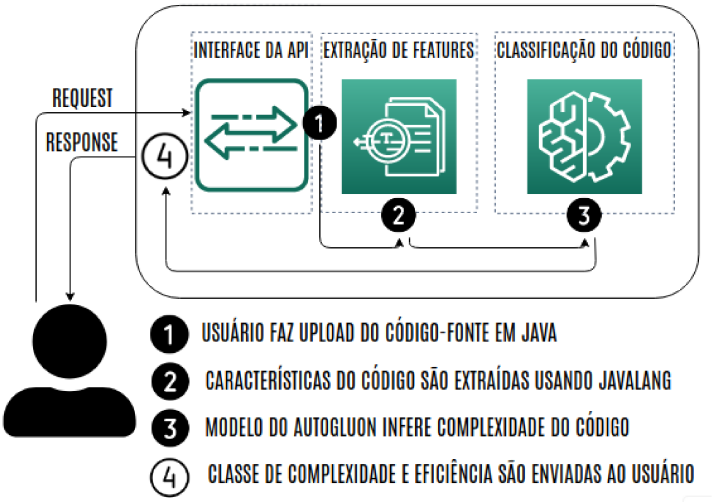


Fonte: Elaborado pelo autor

No canto superior direito da Figura 8, encontra-se o tamanho em bytes da resposta e o tempo de resposta da API para esse POST. Embora esse tempo não seja precisamente o gasto pela API, visto que leva em conta outros processos do Postman, ele proporciona uma estimativa útil da velocidade de inferência da aplicação. Essa métrica é necessária na análise dos resultados para embasar a escolha do modelo final de classificação usado pela API.

De forma geral, o funcionamento da API pode ser exemplificado através do esquema da Figura 9.

Figura 9 – Esquema de funcionamento da API



Fonte: Elaborado pelo autor

A Figura 9 apresenta um resumo dos passos de uma interação entre usuário e aplicação. O usuário faz um POST, em que seu request é feito pelo upload de seu código-fonte. A API extrai as características do código usando Javalang e gera um arquivo csv no padrão do banco de dados Mergeds contendo as features extraídas. O modelo do AutoGluon, que foi treinado no Mergeds dataset, infere a eficiência e a classe de complexidade do código a partir do arquivo csv. Finalmente, a API gera um response contendo essas duas métricas no formato apresentado na Figura 8.

Para a execução desse processo, inúmeros códigos são necessários. Toda programação realizada para o desenvolvimento da API RTCC e deployment da mesma no Heroku está disponível no repositório RTCC, no endereço https://github.com/allesbraun/RTCC/tree/master.

Nesse capítulo foi determinado o funcionamento da maior parte dos componentes da API, como a extração de características e o banco de dados utilizado. Resta, portanto, decidir qual modelo do AutoGluon será utilizado e avaliar o funcionamento da API de forma geral, além de verificar se a mesma cumpriu ou não com os requisitos e objetivos estabelecidos previamente.

4 RESULTADOS E DISCUSSÕES

Este capítulo aborda os resultados obtidos pelos modelos e pela API Rest, sobretudo quanto à acurácia dos modelos e o tempo de resposta dos mesmos na aplicação. Além disso, avalia-se o cumprimento dos requisitos funcionais e não funcionais estipulados no Capítulo 3. Sendo assim, primeiro trata-se dos resultados dos modelos aplicados aos datasets Mergeds, Crawleds e Reference. Após isso, os tempos de resposta dos presets são analisados em busca do melhor custo-benefício para API.

4.1 ANÁLISE DA ACURÁCIA DOS MODELOS

Esta seção apresenta dois cenários de análise. O Cenário 1 aborda o treinamento e teste no Mergeds dataset, que por ser o mais completo dos datasets formados vai ser o utilizado pela API. Enquanto isso, o Cenário 2 trata do treinamento no Crawleds e teste no Reference, com o objetivo de validar o poder de generalização.

4.1.1 Cenário 1: Modelo aplicado ao Mergeds dataset

Formado pela união do Crawleds dataset e do Reference dataset, o Mergeds dataset possui 1668 amostras. A Tabela 10 apresenta a acurácia dos presets treinados nesse banco de dados. A acurácia para efficiency foi apenas 2,15% inferior à do artigo *Estimating Code Running Time Complexity with Machine Learning* (2023), enquanto a acurácia para complexity_class foi cerca de 10% menor. Com relação ao trabalho de Sikka *et al.* (2020), esse cenário foi superior em cerca de 6,5% no que tange acurácia para complexity_class.

Tabela 10 – Acurácia obtida treinando e testando com o Mergeds dataset

Preset	Acurácia para complexity_class	Acurácia para efficiency
`medium_quality`	78,38%	88,02%
`good_quality`	78,37%	86,93%
`high_quality`	78,8%	86,93%
`best_quality`	79,05%	86,93%

Fonte: Elaborado pelo autor.

Com base nos dados da Tabela 10, existem duas conclusões importantes a se fazer. Primeiro, é interessante notar como mesmo com três classes a mais de complexidade (exponencial, polinomial e sublinear), os resultados foram superiores aos de Sikka *et al.* (2020). Além disso, os resultados obtidos por todos os presets cumprem tanto o **RNF1** quanto o **RNF2**. Enquanto o **RNF1** exigia ao menos 80% de acurácia

para efficiency, valor que foi superado por no mínimo 6,94% pelos presets, o **RNF2** buscava 70% de acurácia para complexity_class, o que foi cumprido pelos presets com no mínimo 8,37% de folga.

4.1.2 Cenário 2: Modelo aplicado ao Crawleds e Reference datasets

Na Subseção 4.1.1, por mais que os requisitos desenvolvidos na metodologia tenham sido cumpridos, é visível que os resultados de Pfitscher *et al.* (2023) ainda foram superiores. Por isso, o Cenário 2 busca avaliar se a superioridade em termos de complexity_class do trabalho de Pfitscher *et al.* não está atrelada a um overfitting dos modelos utilizados.

Na Tabela 11 é feita uma análise do AutoGluon treinando com o Crawleds e testando com o Reference dataset, banco de dados de Sikka *et al.* (2020). Com isso, é possível avaliar o poder de generalização do modelo, pois ele não teve contato com o Reference dataset durante o treinamento.

Tabela 11 – Acurácia obtida treinando com o Crawleds e testando com o Reference dataset

Preset	Acurácia para complexity	Acurácia para efficiency
`medium_quality`	68,53%	84,96%
`good_quality`	73,15%	86,36%
`high_quality`	73,15%	86,36%
`best_quality`	73,9%	86,57%

Fonte: Elaborado pelo autor.

É possível observar na Tabela 11 como a acurácia aumentou em relação aos resultados de Pfitscher *et al.* (2023). Enquanto o artigo de Pfitscher *et al.* havia obtido apenas 44,04% ao treinar com o Crawled e testar com o Reference dataset, a inclusão do Augustives dataset no banco de dados de treinamento gerou um salto de 29,86% na acurácia do `[best_quality]`, que agora é de 73,9%. Além disso, a superioridade do AutoGluon para generalização já havia sido demonstrada na comparação apenas entre métodos de classificação, onde o Autogluon havia superado em 16% a acurácia da classe de complexidade do Random Forest.

A partir desses cenários, essa seção torna evidente a melhora tanto em questão de desempenho quanto generalização do modelo. O uso de ensembles criados por técnicas de stacking e bagging feito pelo AutoGluon se apresenta como uma solução mais adequada para identificar as nuances de problemas multiclasse sem relações óbvias entre as features e as classes.

Entretanto, enquanto os requisitos referentes à acurácia dos modelos foram cumpridos, um classificador usado para aplicações em tempo real precisa levar em

consideração seu tempo de resposta, como preza o **RNF3**. Por isso, os tempos obtidos a partir dos testes no Postman, tanto no ambiente de desenvolvimento quanto de produção são relevantes para a montagem final da API.

4.2 ANÁLISE DO TEMPO DE RESPOSTA DOS MODELOS

Esta seção avalia a latência da API para geração da resposta ao cliente. No contexto de tempo de resposta de uma aplicação, trata-se do tempo necessário para uma solicitação ser enviada de um usuário para o servidor, ser processada por ele e a resposta gerada por esse processo ser enviada de volta ao usuário.

Inúmeras variáveis interferem no tempo de resposta da API, como o número de processos internos da API e potenciais overheads da rede. No caso deste trabalho, os processos de classificação de classe de complexidade de tempo de execução e eficiência são os aspectos mais custosos em termos de latência da aplicação. Por isso, existe a necessidade de comparar os presets nesse âmbito.

Para avaliação das métricas de tempo de resposta da API, foi utilizado o Postman, que gera automaticamente com a Response o tempo de resposta. Inicialmente, os testes foram feitos utilizando o servidor local gerado pelo Django. Para uma base mais confiável de dados, foram feitas 31 inferências. Entretanto, tendo em vista a tendência do Postman de obter uma resposta mais lenta após a primeira requisição POST, o primeiro tempo de resposta de cada preset foi descartado. Sendo assim, após a coleta das 30 amostras restantes, foram calculadas suas médias amostrais (Equação 4) e seu desvio-padrão amostral (Equação 5).

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4}$$

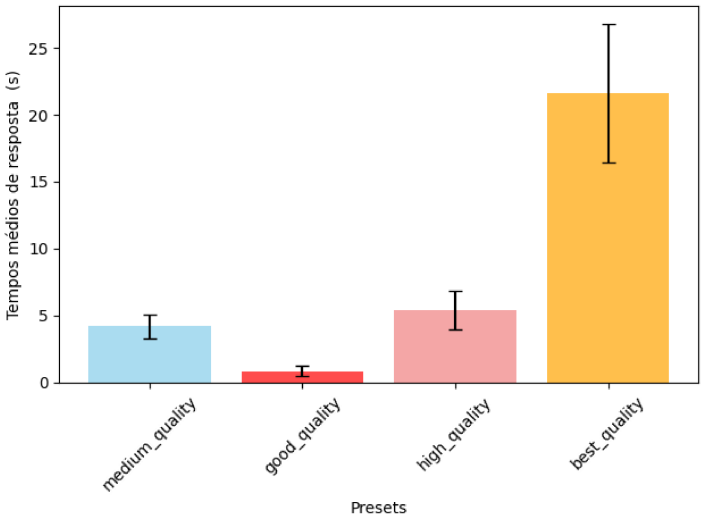
$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{5}$$

Inicialmente, são avaliados na Subseção 4.2.1 os presets em suas configurações mais básicas, que apenas variam o grau do parâmetro `_quality` utilizado. Após isso, são comparadas na Subseção 4.2.2 as configurações com esses mesmos parâmetros de `_quality`, mas acompanhados do preset `[optimize_for_deployment]`, que visa diminuir a latência do processo de geração da Response da API.

4.2.1 Análise dos tempos de resposta dos presets padrão

Por presets padrão, entende-se os quatro presets de qualidade `[medium_quality]`, `[good_quality]`, `[high_quality]` e `[best_quality]`. A Figura 10 compara os tempos de resposta de cada um deles para um mesmo código-fonte de teste.

Figura 10 – Tempo médio de resposta em segundos dos presets com desvio-padrão



Fonte: Elaborado pelo autor.

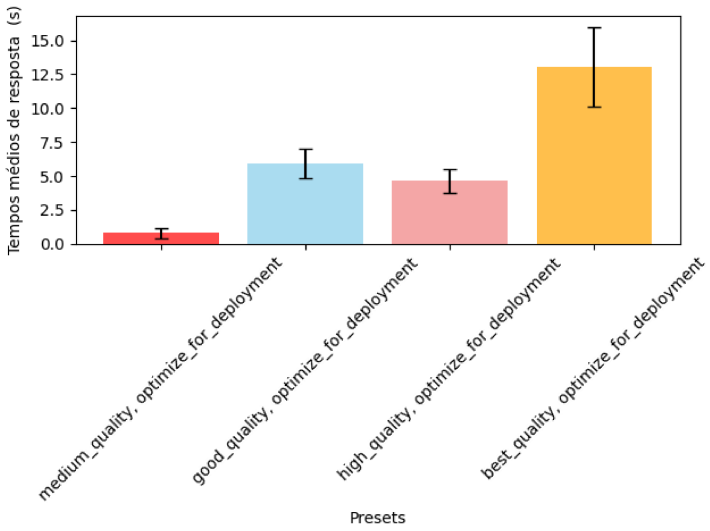
Dentre as configurações comparadas na Figura 10, a de melhor desempenho foi a `[good_quality]`. Ela obteve uma média amostral de 838,7 milissegundos e um desvio-padrão amostral de 377,828 milissegundos.

O requisito não funcional **RNF3** definia que um tempo de resposta aceitável devia ser inferior a dois segundos. Sendo assim, a única configuração que cumpriu o requisito foi a `[good_quality]`, que atingiu uma média inferior a um segundo, além de possuir a menor variação de resultados.

4.2.2 Análise dos tempos de resposta dos presets otimizados

Esta subseção avalia o impacto do acréscimo do preset `[optmize_for_deployment]` no tempo de resposta da API. A Figura 11 inclui as versões dos modelos com o preset, que otimiza o uso de recursos mas pode interferir no desempenho dos modelos (AutoGluon, 2022).

Figura 11 – Tempo médio de resposta em segundos dos presets otimizados com desvio-padrão



Fonte: Elaborado pelo autor.

Para os presets otimizados para deployment comparados na Figura 11, a configuração mais veloz foi a `[medium_quality, optimize_for_deployment]`, que teve uma média amostral de 784.9667 milissegundos e desvio-padrão amostral de 406.025 milissegundos. Novamente, foi o único preset de seu conjunto a obter média inferior a um segundo e a cumprir com o requisito **RNF3**.

Como pôde ser observado nas Figuras 10 e 11, alguns presets otimizados para o deploy obtiveram tempos de resposta maiores que os de suas versões não otimizadas. Isso é justificável, pois algumas vezes as técnicas de otimização provocam falhas durante a inferência. Também é visível como um preset mais complexo como o `[good_quality]` obteve desempenho consideravelmente superior ao do `[medium_quality]`, que é mais simples e, portanto, teoricamente mais rápido. Nesse caso, é preciso lembrar que assim como a acurácia não é sempre proporcional ao número de parâmetros dos presets, o tempo de resposta também não é.

Esta seção concluiu que ambas as configurações `[medium_quality, optimize_for_deployment]` e `[good_quality]` cumprem com o requisito de desempenho estabelecido anteriormente, o **RNF3**. Entretanto, como os dois possuem tempos próximos tanto em média quanto desvio-padrão, com uma pequena vantagem para o `[medium_quality, optimize_for_deployment]`, os dois serão comparados novamente para a escolha do modelo final da API.

4.3 ESCOLHA DO MODELO DE CLASSIFICAÇÃO UTILIZADO NA API

Tendo em vista a proximidade de desempenho dos presets `[medium_quality, optimize_for_deployment]` e `[good_quality]`, esta seção compara o desempenho dos dois em termos de acurácia na inferência, pois muitas vezes a otimização para deployment prejudica a performance do modelo. Sendo assim, é necessário comparar as métricas de `[medium_quality, optimize_for_deployment]` com as descritas para o preset `[good_quality]` na Tabela 10, disponível no Cenário 1 descrito na Seção 4.1. Essa comparação está disponível na Tabela 12.

Tabela 12 – Comparação entre acurácias das duas melhores configurações de presets

Preset	Classe de complexidade	Eficiência
`medium_quality, optimize_for_deployment`	78,08%	87,72%
`good_quality`	78,37%	86,93%

Fonte: Elaborado pelo autor.

Com base nos resultados apresentados na Tabela 12, conclui-se que a melhor configuração para a API RTTC é a `[medium_quality, optimize_for_deployment]`. Com relação aos tempos de resposta, ela foi 6,41% mais veloz que a média do `[good_quality]` para um desvio-padrão apenas 7,46% maior. Enquanto isso, ela é também 0,91% superior na predição de eficiência e apenas 0,37% inferior na predição de classe de complexidade.

Outro fator a ser considerado é que as bibliotecas necessárias para seu funcionamento ocupam menos espaço na memória, o que é relevante ao hospedar a API em um servidor como o Heroku. Como o pacote mais básico de hospedagem nesse serviço possui um limite de 500 MB por projeto, a busca pela otimização dos custos com memória é imprescindível.

Levando em consideração a ocupação de memória e impacto na latência da API, esta seção definiu o `[medium_quality, optimize_for_deployment]` como preset a ser utilizado no ambiente de produção, que neste caso é o Heroku.

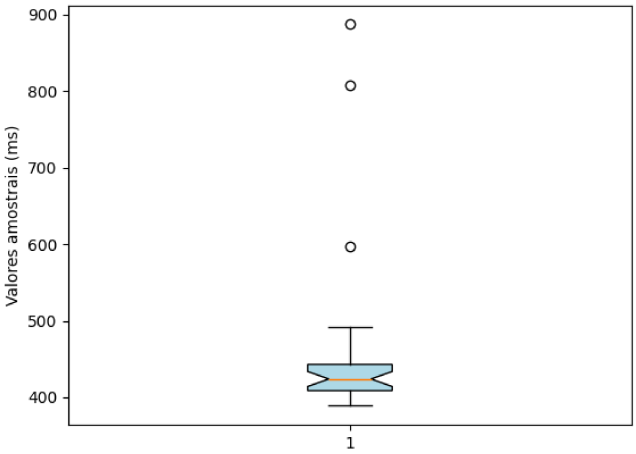
4.4 ANÁLISE DO MODELO ESCOLHIDO PARA A API NO AMBIENTE DE PRODUÇÃO

A partir da definição do preset do AutoGluon escolhido para a API, é preciso analisar o desempenho dessa configuração na API no próprio Heroku, com o intuito de avaliar o impacto de variáveis como conexão com a internet e desempenho do servidor no tempo de resposta da aplicação.

Seguindo com a metodologia implementada anteriormente, foram feitas 31 inferências utilizando o `[medium_quality, optimize_for_deployment]` como preset e

o Mergeds dataset como banco de dados de treinamento. Dessas 31 inferências, a primeira foi novamente descartada e as restantes geraram os resultados que constam na Figura 12.

Figura 12 – Boxplot dos tempos de resposta da API RTTC no Heroku



Fonte: Elaborado pelo autor.

O desempenho mostrado na Figura 12 é superior ao apresentado nos testes no LocalHost. A média amostral de tempo de resposta foi de 458.167 milissegundos e o desvio-padrão amostral foi de 113.488 milissegundos. Esses valores representam uma queda de aproximadamente 41.62% e 72.03% para a média e desvio-padrão amostrais coletados no LocalHost, que eram de 784.967 milissegundos e 406.025 milissegundos, respectivamente.

É contraintuitivo pensar que um ambiente teoricamente mais controlado como o LocalHost apresente uma variação tão significativa de resultados. Entretanto, o foco da análise não é a comparação entre os ambientes, e sim se a latência aceitável dos mesmos se manteve. Além disso, mesmo que essas amostras não representem o desempenho real da API, seus valores baixos indicam que mesmo que na realidade ela seja consideravelmente mais lenta, seu desempenho ainda seja aceitável para o usuário final.

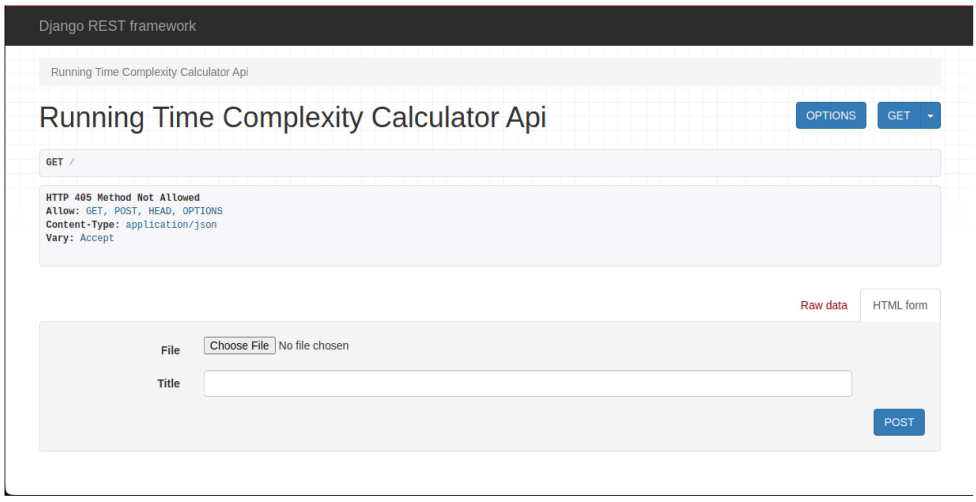
Esta seção mostra que mesmo com a mudança do ambiente de desenvolvimento para o de produção, a API continua cumprindo com o requisito não funcional **RNF3**. Além disso, a média de 0,458 segundos representa apenas 22,9% da limitação de 2 segundos imposta pelo requisito. Essa métrica, aliada a um desvio-padrão amostral inferior a 6% do tempo limite, proporciona um bom nível de confiabilidade para a aplicação.

4.5 DESEMPENHO DA API NO AMBIENTE DE PRODUÇÃO

Nesta seção é exposta a API em execução no Heroku. São mostrados casos teste que exemplificam seu funcionamento perante as entradas, desde o retorno esperado para um código em Java até o retorno para casos de erro como extensão inválida e tamanho acima do limite. Além disso, é visto como o sistema se protege do acesso de pessoas não autorizadas.

Na Figura 13, consta a tela inicial vista pela usuário ao acessar a API. Ele tem a opção de fazer o upload de um arquivo e adicionar um nome a esse. O título não é obrigatório para o envio da requisição POST.

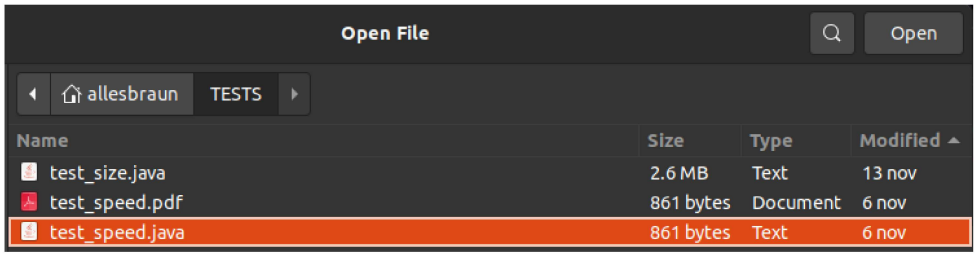
Figura 13 – Interface da API RTTC no Heroku



Fonte: Elaborado pelo autor.

Ao clicar em "Choose file"(Figura 13), uma janela é aberta para que o usuário selecione o arquivo de sua preferência, tal processo pode ser visualizado na Figura 14. No cenário descrito na Figura 15, o usuário seleciona o código-fonte usado nos testes de velocidade de tempo de resposta da API. Portanto, espera-se que o POST desse código gere uma resposta para a classe de complexidade e eficiência.

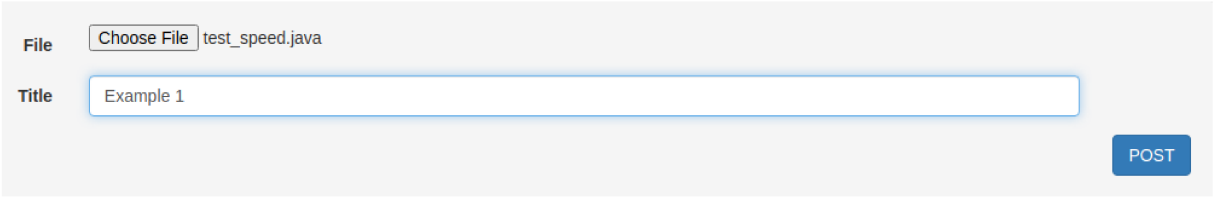
Figura 14 – Seleção do arquivo test_speed.java



Fonte: Elaborado pelo autor.

O upload bem-sucedido que ocorre nas Figuras 14 e 15 do arquivo mostra o cumprimento do requisito funcional **RF2**.

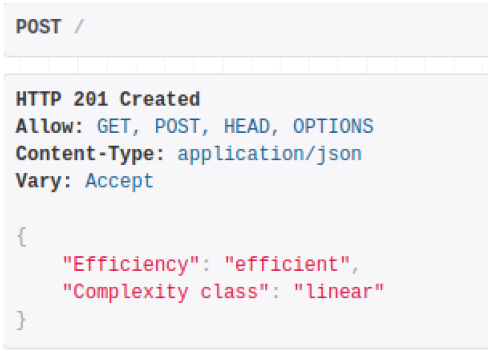
Figura 15 – Arquivo test_speed.java adicionado no formulário de envio



Fonte: Elaborado pelo autor.

Conforme esperado, a Figura 16 apresenta os resultados referentes à classe de complexidade e eficiência do código-fonte test_speed.java. Para esse exemplo, a API primeiro classificou o código-fonte com base na eficiência, o intitulando como eficiente. Após isso, o definiu como pertencente à classe de complexidade linear, o que cumpre **RF1**. Além disso, independente das inferências feitas estarem corretas, a Response foi legível e compreensível, cumprindo o requisito **RNF8**.

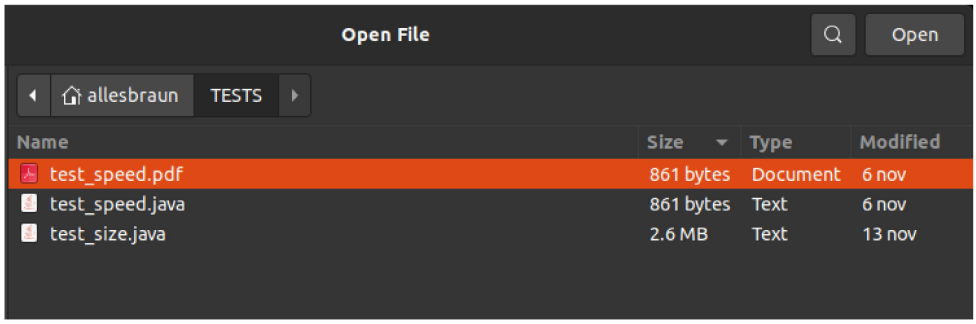
Figura 16 – Response da API para test_speed.java



Fonte: Elaborado pelo autor.

Na Figura 17, busca-se analisar como a API reage a um upload de um arquivo que não é uma extensão java. Nesse exemplo, é feito o upload de um arquivo pdf, o que deve resultar em uma Response alertando o erro.

Figura 17 – Seleção do test_speed.pdf



Fonte: Elaborado pelo autor.

Como pode ser visto na Figura 18, a API consegue identificar que não se trata de um código-fonte em Java. Por isso, ela gera uma resposta de alerta dizendo que o arquivo deve ter extensão .java. Além disso, a Response apresenta o status HTTP 400 Bad Request, que indica que o servidor não pode ou não consegue processar a requisição devido a um erro gerado pelo cliente, neste caso, a extensão inválida.

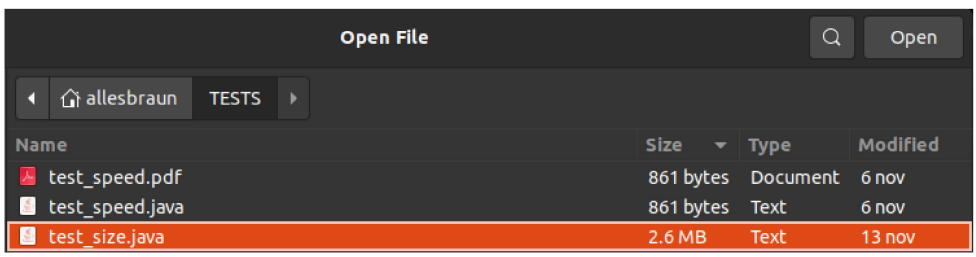
Figura 18 – Response da API para test_speed.pdf

```
POST /

HTTP 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "error": "The file must have a .java extension"
}
```

Fonte: Elaborado pelo autor.

Outro limitante do upload do usuário é o selecionado na Figura 19. Para combater ataques DoS, o sistema limita o tamanho do arquivo em Java a 1 MB. Como pode ser visto, a figura apresenta um arquivo de 2,6 Megabytes.

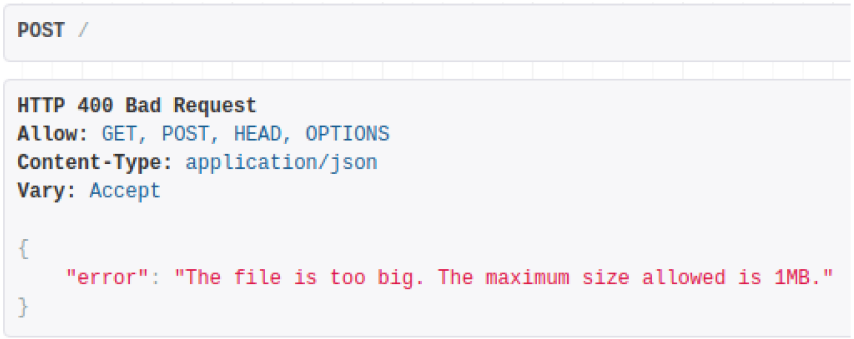
Figura 19 – Seleção do test_size.java



Fonte: Elaborado pelo autor.

A Figura 20 mostra que o sistema identificou um arquivo de tamanho maior que o permite e gerou corretamente a mensagem de erro indicando que o arquivo é grande demais e que o limite é 1 MB, além de também gerar o status 400 Bad Request.

Figura 20 – Response da API para test_size.java



Fonte: Elaborado pelo autor.

A partir dos três exemplos desenvolvidos nessa seção, é possível concluir que a API responde corretamente a envios do usuário, tanto gerando o Response no formato correto quanto obedecendo às restrições impostas acerca do tamanho e tipo do arquivo. Com isso, os requisitos funcionais **RF2** e os requisitos não funcionais **RNF5**, **RNF6**, **RNF7** e **RNF8** podem ser considerados cumpridos.

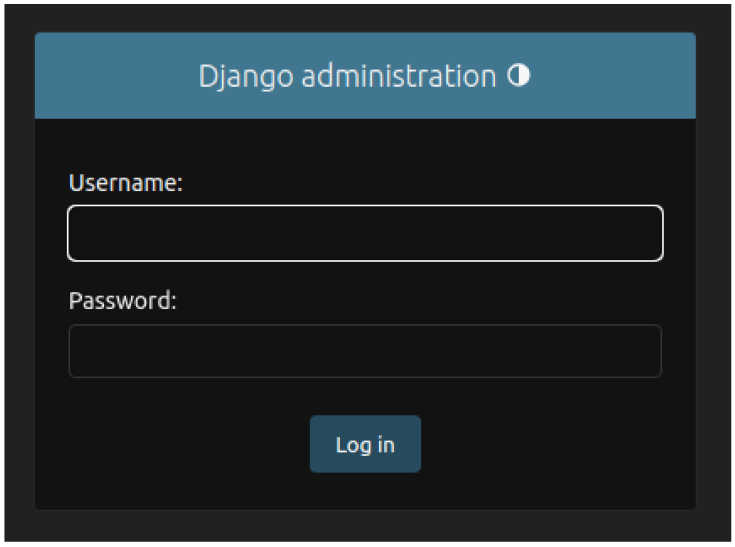
4.6 ACESSO AUTORIZADO E NÃO AUTORIZADO NA API

Além da proteção contra ataques DoS, outro elemento importante da segurança do sistema é garantir que usuários não autorizados não obtenham acesso à áreas dos desenvolvedores, como a página de administrador do Django. Por isso, esta seção investiga como é o acesso à essa página e se os requisitos de segurança são obedecidos.

Toda API desenvolvida no Django possui, por padrão, uma URI (Uniform Resource Identifier) admin. Essa página é a interface de administração automática onde o administrador da aplicação acessa funcionalidades especiais. Ela lê os metadados de seu Model para fornecer uma interface pronta para produção (Foundation; contributors, 2023).

Na Figura 21, é possível observar a tela de login da administração. Uma das maneiras de acessá-la é adicionando ao lado da URL inicial do sistema, neste caso `https://rtcc-40c886516567.herokuapp.com`, o endpoint `/admin`.

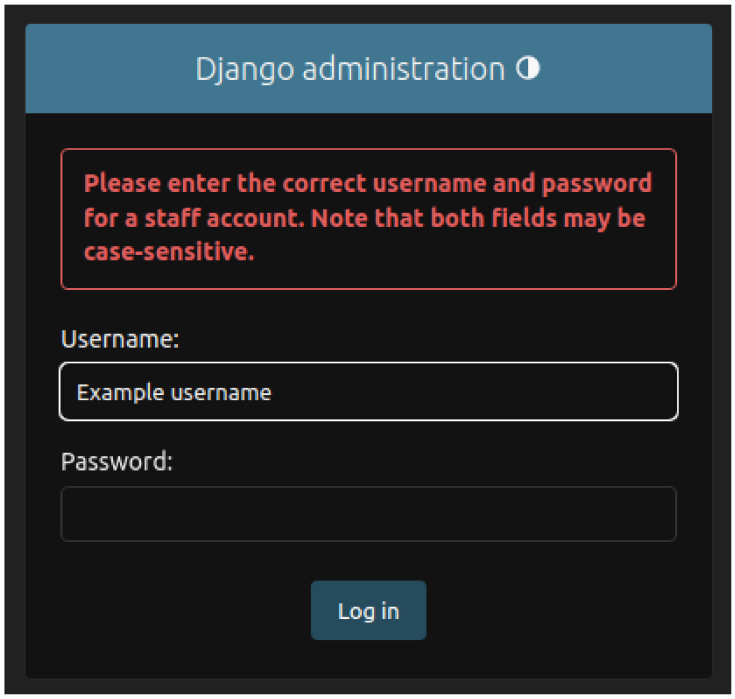
Figura 21 – Tela de login da administração



Fonte: Elaborado pelo autor.

Inicialmente, o formulário de login foi preenchido com usuário e senha fictícios. Conforme pode ser visto na Figura 22, o acesso foi negado como era esperado. Com isso, o requisito **RNF10** foi cumprido.

Figura 22 – Acesso negado para usuário não cadastrado como administrador



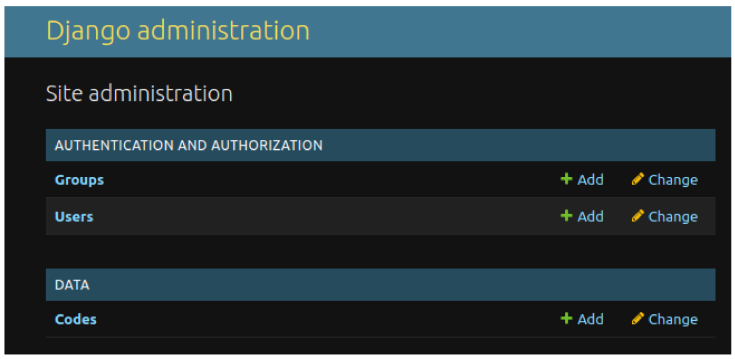
Fonte: Elaborado pelo autor.

Na tentativa seguinte, foram utilizados o usuário e senha cadastrados no projeto Django por meio do comando `createsuperuser`. Esse comando cria uma conta de usuário

que tem todas as permissões, e é, portanto, capaz de acessar a página de admin (Foundation; contributors, 2023).

Como esperado, o login foi feito com sucesso, o que cumpre o requisito **RF4**. Sendo assim, foi possível acessar a página de administrador da API, mostrada na Figura 23. Nessa página, é possível adicionar e editar o relacionamento de grupos e usuários com a API, desde dar a um usuário os recursos de administrador como gerenciar os métodos HTTP que ele é capaz de utilizar.

Figura 23 – Recursos disponíveis para o administrador



Fonte: Elaborado pelo autor.

Nessa mesma Figura 23, existe o link `Codes`, que leva à página que permite que o admin adicione ou edite códigos já existentes no banco da API. Entretanto, nem todas as funcionalidades nesse âmbito estão disponíveis no momento para a API, o que é melhor tratado na Seção 4.7.

Após o teste do login do administrador, é possui concluir que a API cumpre os requisitos **RNF10** e **RF4** de segurança. Entretanto, para que ela se torne eficaz tanto na análise de complexidade como no seu funcionamento como aplicação, ela precisa superar limitações atuais e se desenvolver para o cumprimento de um conjunto mais amplo de requisitos.

4.7 LIMITAÇÕES

Essa seção visa lidar com as limitações presentes na aplicação atualmente. Embora os recursos atuais já cumpram todos os requisitos funcionais testados até o momento, a busca por um refino de suas capacidades é fundamental.

A limitação que impacta os objetivos de médio prazo deste projeto de forma mais imediata é o não armazenamento das informações dos códigos-fonte enviados pelos usuários da API. Embora essa funcionalidade estivesse disponível no ambiente de desenvolvimento da API, seu uso no ambiente de produção sem o estudo aprofundado das regras da Lei Geral de Proteção de Dados resultaria no não cumprimento do

requisito não funcional **RNF9**. Portanto, enquanto essas regras não forem cumpridas e essa limitação existir, os códigos recebidos não poderão ser convertidos em material para o banco de dados da API.

Além da limitação de expansão do banco de dados, a limitação dos testes aplicados à API dificulta uma avaliação mais precisa do desempenho da mesma. Testes envolvendo um alto número de requisições são necessários para traçar o desempenho da aplicação no mundo real. A utilização de outro API auxiliar como o Postman, ou mesmo o uso de funções pagas dentro dessa aplicação podem ajudar na resolução desse problema. Além disso, a limitação de espaço de memória imposta pelo Heroku restringe a acurácia da aplicação, visto que impede o uso de bibliotecas de modelos mais potentes.

Com relação aos requisitos estabelecidos na metodologia, o Quadro 2 mostra que o projeto cumpriu todos os requisitos funcionais.

Quadro 2 – Cumprimento dos requisitos funcionais

Requisito	**Temática**	**Status**
RF1	Resposta da API	Cumprido
RF2	Tratamento de arquivos	Cumprido
RF3	Extração de características	Cumprido
RF4	Controle de acesso	Cumprido

Embora o Quadro 2 apresente o cumprimento do requisito **RF3**, que requeria que a API fosse capaz de extrair características dos códigos recebidos, conforme foi visto no Capítulo 3, a extração atingiu 94,2% de taxa de acerto. Embora a diferença entre os resultados obtidos e o resultado ideal (100%) seja de apenas 5,8%, a ideia de que aproximadamente cinco em cada 100 pessoas que submetam seus códigos para análise não tenham seus códigos avaliados corretamente é aceitável apenas em uma fase inicial de desenvolvimento, como é o caso deste trabalho. Por isso, a extração de características de código é a limitação que deve ser otimizada com maior urgência.

Com relação aos requisitos não funcionais, o Quadro 3 mostra que o projeto cumpriu com o planejamento para todas as áreas, o que destaca seu desempenho promissor.

Quadro 3 – Cumprimento dos requisitos não funcionais

Requisito	**Temática**	**Status**
RNF1	Acurácia de classificação	Cumprido
RNF2	Acurácia de classificação	Cumprido
RNF3	Resposta da API	Cumprido
RNF4	Limitação do Heroku	Cumprido
RNF5	Tratamento de arquivos	Cumprido
RNF6	Tratamento de arquivos	Cumprido
RNF7	Resposta da API	Cumprido
RNF8	Resposta da API	Cumprido
RNF9	Segurança de dados	Cumprido
RNF10	Controle de acesso	Cumprido

Embora todos os requisitos não funcionais apresentados no Quadro 3 tenham sido cumpridos, a presença do **RNF9** acaba limitando as funcionalidades atuais da API. Sendo assim, esta é a segunda limitação mais urgente, pois impede o armazenamento de dados para expansão do banco de dados da API.

Este capítulo analisou o resultados de diversas configurações de presets do AutoGluon nos datasets Crawleds, Mergeds e Reference. Além da análise da acurácia, houve um foco do tempo de resposta de API para cada modelo. Com base nisso, foi definido que o modelo com a melhor relação entre acurácia e latência é o `[medium_quality, optimize_for_deployment]`.

A configuração `[medium_quality, optimize_for_deployment]` foi utilizada na API no ambiente de produção, onde o RTCC foi hospedado no serviço Heroku. O desempenho da aplicação nesse ambiente foi analisado com base no requisitos estipulados na metodologia, com resultados satisfatórios para todos os **RNF** e **RF**. Embora os resultados do projeto possam ser considerados positivos, a API ainda não pode ser considerada pronta para ser disponibilizada ao público.

5 CONCLUSÕES

Este Trabalho de Conclusão de Curso buscou aprimorar a análise de eficiência de algoritmos por meio do desenvolvimento de uma API RESTful denominada RTCC (Running Time Complexity Calculator). Durante o desenvolvimento do projeto, foram definidos objetivos que tangenciaram a elaboração da API RTCC. O principal deles era que a aplicação fosse capaz de estimar a classe de complexidade de um código-fonte em Java. Para o cumprimento desse objetivo, foram determinadas metas secundárias relacionadas à partes fundamentais do funcionamento da API, como o desenvolvimento de um banco de dados mais robusto, a implementação de um método de extração de características de código eficiente e a elaboração de um método de classificação superior aos utilizados nos trabalhos relacionados.

Ao revisitar estudos anteriores, constatou-se a importância de um banco de dados representativo e balanceado. O acréscimo de amostras do dataset Augustives aos bancos de dados utilizados por Pfitscher *et al.* (2023) foi fundamental para superar as limitações observadas em trabalhos anteriores no que diz respeito à capacidade de generalização dos modelos. A junção desses bancos de dados gerou um banco chamado Mergeds dataset, com 1668 amostras divididas em oito classes de complexidade.

Além do novo banco de dados, a utilização do framework de AutoML AutoGluon para classificação das classes de complexidade proporcionou tanto um tratamento eficaz do banco de dados quanto uma comparação mais eficiente de modelos, que puderam ser testados em maior variedade. No entanto, a maior vantagem do uso de AutoGluon foi a utilização de seus métodos de ensemble, que através da combinação de diversas técnicas de stacking e bagging geraram resultados significativos de acurácia na classificação. Por meio desses métodos, foi possível obter um máximo de 79,05% de acurácia para classe de complexidade e de 88,02% para eficiência.

Para a criação das features empregadas nos modelos de classificação do AutoGluon, foi necessário elaborar um método de extração de características de código. Para isso, utilizou-se a biblioteca Javalang, que capta as informações a partir de uma análise da AST do código-fonte. Essa biblioteca foi suficiente visto que a API busca apenas lidar com códigos em Java, já que é a única linguagem presente nos bancos de dados dos trabalhos relacionados a este projeto. A extração de características desenvolvida obteve uma taxa de assertividade de 94,2%, o que indica que a implementação atual do Javalang falha em determinados momentos.

Com relação à API, o projeto e desenvolvimento foi feito utilizando as funcionalidades dos frameworks Django e Django Rest Frameworks aplicados à arquitetura REST. Embora apenas o método POST tenha sido implementado até o

momento, a arquitetura e frameworks usados propiciam um desenvolvimento facilitado de novas funcionalidades para a aplicação.

É possível resumir as contribuições deste projeto em três pontos:

- Consolidação de um banco de dados mais extenso e representativo do que os disponíveis atualmente para pesquisas na área.
- Introdução e comprovação da eficácia do uso de AutoML, sobretudo o AutoGluon, em pesquisas relacionadas à classificação de classe de complexidade de tempo de execução a partir de características de código.
- Desenvolvimento de uma aplicação RESTful para estimação de classe de complexidade.

Tendo em vista que a API ainda é limitada em alguns aspectos que a privam de um acesso público, as duas maiores contribuições deste trabalho ainda devem ser implementadas: a possibilidade de desenvolvedores e pesquisadores enviarem seus códigos e obterem instantaneamente informações sobre sua complexidade e a geração automatizada de novas amostras para o banco de dados atual a partir dos uploads feitos por usuários.

Sendo assim, os próximos passos dessa pesquisa devem ser a implementação das regras da LGPD, para que a API possa ser disponibilizada ao público, e a correção das falhas na extração de características. Seria interessante substituir o uso do Javalang por alguma ferramenta de reconhecimento de linguagem, como o ANTLR4 (The ANTLR Project, 2023), pois além do Javalang ter apresentado erros durante a extração de características, outra ferramenta mais completa proporcionaria a extração de características em qualquer linguagem.

A variedade de linguagens aceitas na API resultaria em um público mais extenso, o que tornaria a API mais útil para a sociedade. Além disso, a maior variedade de códigos tende a produzir um banco de dados mais completo. Outro ponto que passa a necessitar de atenção é o estabelecimento de um mecanismo de validação dos códigos obtidos para que possam ser adicionados aos dados de treinamento já existentes com uma alta confiabilidade.

Além dos dados, existe um grande escopo de métodos de classificação e abordagens não testados. Uma possível variação do trabalho atual seria substituir a abordagem global multiclasse utilizada para uma abordagem híbrida composta por partições tanto locais quanto globais. Estudos recentes indicam que isso pode aumentar significativamente a capacidade do modelo de encontrar correlações entre rótulos, o que seria particularmente útil para diferenciar algoritmos com classes de complexidade vizinhas, como a linear e logarítmica (Gatto *et al.*, 2023).

É fundamental também continuar desenvolvendo a API, que ainda se encontra em um estágio inicial. Técnicas para otimização de performance, como a configuração eficiente de cache, aliadas a um banco de dados mais robusto do que SQLite atual são

importantes a longo prazo, tendo em vista um aumento na demanda da aplicação. Além disso, práticas de integração e entrega contínuas proporcionariam um desenvolvimento mais qualificado do projeto. O mesmo pode ser dito de testes de integração, que representariam um salto na confiabilidade dos resultados obtidos.

Com base nos resultados positivos e nas limitações apresentadas, pode-se concluir que a API RTCC representa um avanço tanto na praticidade da complexidade de tempo de execução de códigos quanto no estudo de como as características impactam na eficiência no algoritmo. Tendo em vista o cumprimento não apenas dos objetivos propostos mas também requisitos estipulados na metodologia, a abordagem apresentada ao longo da pesquisa pode ser considerada bem-sucedida. Além disso, embora os resultados não sejam definitivos e a aplicação necessite de um período mais longo de desenvolvimento para ser utilizada pelo público, seu impacto nas áreas de AutoML e complexidade de algoritmo pode ser considerado relevante.

REFERÊNCIAS

AUDIBERT, R. B. *et al.* **On the Evolution of A.I. and Machine Learning: *Towards Measuring and Understanding Impact, Influence, and Leadership at Premier A.I. Conferences***. 2022.

AUTOGLUON. **Predicting Columns in a Table - In Depth¶**. 2022. Auto.gluon.ai/. Disponível em: https://auto.gluon.ai/0.4.0/tutorials/tabular_prediction/tabular-indepth.html#sec-tabularadvanced.

AUTOGLUON. **AutoGluon Tabular - Essential Functionality**. 2023. https://auto.gluon.ai/stable/tutorials/tabular/tabular-essentials.html#presets. Acesso em: 01 de outubro de 2023.

BIAU, G.; SCORNET, E. A random forest guided tour. **Test**, Springer, v. 25, p. 197–227, 2016.

BOSSAN, B.; FEIGL, J.; KAN, W. **Otto Group Product Classification Challenge and BNP Paribas Cardif Claims Management**. 2015, 2016. Www.kaggle.com. Disponível em: https://www.kaggle.com/c/otto-group-product-classification-challenge/leaderboard (Otto Group Product Classification Challenge) e https://www.kaggle.com/competitions/bnp-paribas-cardif-claims-management/leaderboard (BNP Paribas Cardif Claims Management).

CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. *In*: **Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining**. San Francisco, USA: ACM, 2016. p. 785–794.

CHIVERS, I. *et al.* An introduction to algorithms and the big o notation. **Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77**, Springer, p. 359–364, 2015.

CORMEN, T. *et al.* **Algoritmos**: Teoria e prátrica. São Paulo: Editora Campos, 2012. v. 3.

DIETTERICH, T. G. Ensemble methods in machine learning. *In*: SPRINGER. **International workshop on multiple classifier systems**. Cambridge, United Kingdom, 2000. p. 1–15.

ERICKSON, N. *et al.* Autogluon-tabular: Robust and accurate automl for structured data. **arXiv preprint arXiv:2003.06505**, 2020.

FEURER, M. *et al.* Efficient and robust automated machine learning. **Advances in neural information processing systems**, v. 28, 2015.

FOUNDATION, D. S.; CONTRIBUTORS. **django-portuguese**. 2023. https://django-portuguese.readthedocs.io/en/1.0/ref/contrib/admin.html. Acesso em: 12 de outubro de 2023.

GAGLIARDI, V. Modern django and the django rest framework. *In*: APRESS. **Decoupled Django : Understand and Build Decoupled Django Architectures for JavaScript**

Front-ends. Berkeley, CA: Apress, 2021. p. 31–40. ISBN 978-1-4842-7144-5. Disponível em: https://doi.org/10.1007/978-1-4842-7144-5_3.

GATTO, E. C. *et al.* Community detection for multi-label classification. *In*: SPRINGER. **Brazilian Conference on Intelligent Systems**. Belo Horizonte, Brasil, 2023. p. 78–93.

GHIMIRE, D. Comparative study on python web frameworks: Flask and django. India, 2020.

GOLMOHAMMADI, A.; ZHANG, M.; ARCURI, A. Testing restful apis: A survey. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, 2022.

HE, X.; ZHAO, K.; CHU, X. Automl: A survey of the state-of-the-art. **Knowledge-Based Systems**, Elsevier, v. 212, p. 106622, 2021.

HUTTER, F.; KOTTHOFF, L.; VANSCHOREN, J. **Automated machine learning**: methods, systems, challenges. The Campus, 4 Crinan Street, London, N1 9XW, United Kingdom: Springer Nature, 2019.

INC, P. **Postman API Platform**. 2023. https://www.postman.com/api-platform/. Acesso em: 26 de outubro de 2023.

JORDAN, M. I.; MITCHELL, T. M. **Machine learning**: Trends, perspectives, and prospects. **Science**, American Association for the Advancement of Science, v. 349, n. 6245, p. 255–260, 2015.

KARMAKER, S. K. *et al.* Automl to date and beyond: Challenges and opportunities. **ACM Computing Surveys (CSUR)**, ACM New York, NY, v. 54, n. 8, p. 1–36, 2021.

KLEINBERG, J.; TARDOS, E. **Algorithm design**. Noida, Uttar Pradesh,India: Pearson Education India, 2006.

KRANE, D. E.; RAYMER, M. L. Analyzing algorithms & asymptotic notation. **Computer Science and Engineering Faculty Publications**, Dayton, OH, USA, 2003.

LUCAS, S. The origins of the halting problem. **Journal of Logical and Algebraic Methods in Programming**, Elsevier, v. 121, p. 100687, 2021.

MENDOZA, H. *et al.* Towards automatically-tuned neural networks. *In*: PMLR. **Workshop on automatic machine learning**. New York, USA, 2016. p. 58–65.

MURPHY, K. P. **Machine learning**: a probabilistic perspective. One Rogers Street, Cambridge, MA 02142-1209, United States: MIT press, 2012.

OLIVEIRA, A. Silva de. **SACI**. 2015. Github.com. Disponível em: https://github.com/Augustives/SACI.

PEREIRA, A.; OLIVEIRA SIMONETTO, E. de. Indústria 4.0: conceitos e perspectivas para o brasil. **Revista da Universidade Vale do Rio Verde**, v. 16, n. 1, 2018.

PFITSCHER, R. J. *et al.* Estimating code running time complexity with machine learning. *In*: SPRINGER. **Brazilian Conference on Intelligent Systems**. Belo Horizonte, Brasil, 2023. p. 400–414.

ROBBES, R.; LUNGU, M.; JANES, A. Api fluency. *In*: **2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)**. Montreal, Canada: IEEE Press, 2019. p. 97–100.

SIKKA, J. *et al.* Learning based methods for code runtime complexity prediction. *In*: **Advances in Information Retrieval: 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14–17, 2020, Proceedings, Part I 42**. 11 W 42nd St Fl 15, New York, NY 10036, United States: Springer, 2020. p. 313–325.

SOLÓRZANO, A. L. V.; CHARÃO, A. S. Explorando a plataforma de computação em nuvem heroku para execução de programas paralelos com openmp. *In*: SBC. **Anais da XVII Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. Porto Alegre, Brasil, 2017.

SOMMERVILLE, I.; SAWYER, P. Viewpoints: principles, problems and a practical approach to requirements engineering. **Annals of software engineering**, Springer, v. 3, n. 1, p. 101–130, 1997.

STRACHEY, C. An impossible program. **The Computer Journal**, The British Computer Society, v. 7, n. 4, p. 313–313, 1965.

SYME, D. *et al.* Lexing and parsing. **Expert F#**, Springer, p. 461–489, 2007.

The ANTLR Project. **ANTLR - Another Tool for Language Recognition**. 2023. https://www.antlr.org/about.html. Acesso em: 13 de maio de 2023.

THUNES, C. **javalang**. 2023. https://github.com/c2nes/javalang. Acesso em: 26 de outubro de 2023.

WALCZAK, S. Artificial neural networks. *In*: **Advanced methodologies and technologies in artificial intelligence, computer simulation, and human-computer interaction**. Pennsylvania, United States: IGI global, 2019. p. 40–53.

WIEGERS, K. E.; BEATTY, J. **Software requirements**. 3. ed. London, United Kingdom: Pearson Education, 2013.

WOLTMANN, S. **Big O notation and time complexity - easily explained**. HappyCoders.eu, 2022. Disponível em: https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/#O1_-_Constant_Time.

ZHAO, B. Web scraping. **Encyclopedia of big data**, Springer Living ed. Cham, p. 1–3, 2017.

ZHOU, L. *et al.* Machine learning on big data: Opportunities and challenges. **Neurocomputing**, Elsevier, v. 237, p. 350–361, 2017.

APÊNDICE A - PRODUÇÃO CIENTÍFICA

Parte da pesquisa apresentada neste trabalho foi reportada à comunidade acadêmica através de um artigo submetido e publicado pelo congresso Brazilian Conference on Inteligent Systems. Os processos de pesquisa, submissão e correção a partir de feedbacks, além da apresentação do artigo em um congresso e do contato com diversos acadêmicos da área, foram fundamentais para o amadurecimento do tema e qualificação desse Trabalho de Conclusão de Curso. Por isso, uma descrição desse artigo consta a seguir.

.1 A.1 ARTIGOS: ACEITOS E EM REVISÃO

1. Ricardo José Pfitscher, Gabriel Braun Rodenbusch, Anderson Dias, Paulo Vieira and Nuno M.M.D. Fouto. **Estimating Code Running Time Complexity with Machine Learning**. 2023 Brazilian Conference on Intelligent Systems (BRACIS). Belo Horizonte, Brasil, 2023 pp. 400-414. DOI: 10.1007/978-3-031-45389-2_27.
    - Título: Estimating Code Running Time Complexity with Machine Learning.
    - Contribuição: Comparação entre novas abordagens de machine learning para classificação de complexidade de tempo de execução e consolidação de banco de dados para classes consideradas ineficientes.
    - Abstract: The running time complexity is a crucial measure for determining the computational efficiency of a given program or algorithm. Depending on the problem complexity class, it can be considered intractable; a program that solves this problem will consume so many resources for a sufficiently large input that it will be unfeasible to execute it. Due to Alan Turing's halting problem, it is impossible to write a program capable of determining the execution time of a given program and, therefore, classifying it according to its complexity class. Despite this limitation, an approximate running time value can be helpful to support development teams in evaluating the efficiency of their produced code. Furthermore, software-integrated development environments (IDEs) could show real-time efficiency indicators for their programmers. Recent research efforts have made, through artificial intelligence techniques, complexity estimations based on code characteristics (e.g., number of nested loops and number of conditional tests). However, there are no databases that relate code characteristics with complexity classes considered inefficient (e.g., $$O(c^n)$$O(cn)and O(n!)), which limits current research results. This research compared three machine learning approaches (i.e., Random Forest, eXtreme Gradient Boosting, and Artificial Neural Networks) regarding their accuracy in predicting Java program codes'

efficiency and complexity class. We train each model using a dataset that merges data from literature and 394 program codes with their respective complexity classes crawled from a publicly available website. Results show that Random Forest resulted in the best accuracy, being 90.17% accurate when predicting codes' efficiency and 89.84% in estimating complexity classes.

- Status: Publicado.
- Qualis: A2.
- Conferência: 12th Brazilian Conference on Intelligent Systems.
- Data: 25-29 de setembro de 2023.
- Local: Belo Horizonte, Brasil.
- URL: <https://link.springer.com/chapter/10.1007/978-3-031-45389-2_27>
- Digital Object Identifier (DOI): <https://doi.org/10.1007/978-3-031-45389-2_27>.

APÊNDICE B - FORMATO DO BANCO DE DADOS

.2 B.1 AUGUSTIVES DATASET: AMOSTRA DO FORMATO DO BANCO DE DADOS

Figura 24 – Primeiras onze colunas do arquivo augustives.csv

num_if	num_else	num_switch	num_loop	num_break	num_Priority	num_binSearch	num_minMax	num_sort	num_hash_map	num_hash_set
2	1	0	0	0	0	0	1	0	0	0
3	1	0	2	0	0	0	1	0	0	0
2	1	0	2	0	0	0	1	0	0	0

Fonte: Elaborado pelo autor

Figura 25 – Últimas oito colunas do arquivo augustives.csv

num_recursive	num_nested_loop	num_vari	num_method	num_state	complexity_class	efficiency	filename
3	0	4	3	6	exponential	inefficient	0-1-knapsack-problem-dp-10-1.java
3	2	7	4	11	quadratic	inefficient	0-1-knapsack-problem-dp-10-2.java
0	2	6	3	8	quadratic	inefficient	0-1-knapsack-problem-dp-10-3.java

Fonte: Elaborado pelo autor