JNT FACIT BUSINESS AND TECHNOLOGY JOURNAL
ISSN: 2526-4281 - ANO 2026 - MÊS DE ABRIL
Ed. 73. Vol. 1 - Págs. 188-206
DOI: 10.5281/zenodo.19570481

CAPES

CNPq
Conselho Nacional de Desenvolvimento Científico e Tecnológico

QUALIS A2

AVALIAÇÃO COMPARATIVA ENTRE OS MODELOS RANDOM FOREST E MLP NA DETECÇÃO DE BUGS[1]

COMPARATIVE EVALUATION BETWEEN THE RANDOM FOREST AND MLP MODELS IN DETECTING BUGS

Filipe Dias BARRETO
Universidade Estadual do Tocantins (UNITINS)
E-mail: filipedias@unitins.br/filipedias25.fd@gmail.com
ORCID: http://orcid.org/0009-0005-6247-7685

Alex COELHO
Universidade Estadual do Tocantins (UNITINS)
E-mail: alex.c@unitins.br/alex.coelho@mail.uft.edu.br
ORCID: http://orcid.org/0000-0002-1576-7242

RESUMO

A detecção automática de defeitos em código-fonte é um desafio central na Garantia de Qualidade de Software, sendo essencial para reduzir custos, aumentar a confiabilidade dos sistemas e agilizar o desenvolvimento. Este trabalho realiza uma avaliação comparativa entre os algoritmos Random Forest e *Multilayer Perceptron* na detecção de bugs em código-fonte. Foram utilizados datasets rotulados, submetidos a etapas de pré-processamento, extração de características e treinamento dos modelos. A avaliação considerou as métricas acurácia, precisão, recall e F1-score. Os resultados indicaram que o Random Forest apresentou melhor desempenho em dados estruturados, enquanto o *Multilayer Perceptron* obteve desempenho superior em dados textuais vetorizados. Conclui-se que não há um modelo universalmente superior, sendo a escolha dependente das características dos dados. O estudo contribui ao fornecer evidências experimentais que auxiliam na seleção de técnicas mais adequadas para a predição de defeitos em software.

Palavras-chave: Detecção de Bugs. Qualidade de Software. Aprendizado Supervisionado. Machine Learning.

[1] COMO CITAR: (ABNT): BARRETO, F. D.; COELHO, A. Avaliação Comparativa entre os Modelos Random Forest e MLP na Detecção de Bugs. **JNT Facit Business and Technology Journal.** Qualis A2. ISSN: 2526-4281, Mês de Abril de 2026 - Ed. 73. VOL. 01. Págs. 188-206. Disponível: http://revistas.faculdadefacit.edu.br. Acesso em: ___/___/___.

ABSTRACT

Automatic source code defect detection is a central challenge in Software Quality Assurance, being essential for reducing costs, increasing system reliability, and speeding up development. This work performs a comparative evaluation between the Random Forest and Multilayer Perceptron algorithms in detecting bugs in source code. Labeled datasets were used, subjected to preprocessing, feature removal, and model training steps. The final evaluation is made with accuracy, precision, recall, and F1-score scores. The results indicated that Random Forest performed better on structured data, while Multilayer Perceptron performed better on vectorized textual data. It is concluded that there is no universally superior model, and the choice depends on the characteristics of the data. The study contributes by providing experimental evidence that assists in selecting the most appropriate techniques for predicting software defects.

Keywords: Bug Detection. Software Quality. Supervised Learning. Machine Learning.

INTRODUÇÃO

A identificação de bugs em códigos-fonte é um tema recorrente em pesquisas sobre software, e a aplicação da Inteligência Artificial (IA) para apoiar tais investigações tem se mostrado cada vez mais promissora. A área de detecção automática de bugs ainda enfrenta desafios importantes, como a falta de padronização experimental, a diversidade de sistemas avaliados e a dificuldade de reproduzir comparações entre diferentes abordagens. Essas limitações são destacadas por estudos recentes que analisam o panorama industrial e acadêmico da predição de defeitos, como destacado em Daza (2025).

Entre as abordagens recentes, destaca-se o uso de redes neurais do tipo Multilayer Perceptron (MLP), um modelo supervisionado amplamente aplicado na predição de defeitos em software. Esse modelo é capaz de aprender padrões complexos presentes nos dados, permitindo a identificação de falhas em código-fonte de forma mais eficiente. Estudos demonstram que abordagens baseadas em aprendizado profundo e redes neurais podem alcançar bons resultados na detecção de bugs, superando métodos tradicionais em determinados cenários e possibilitando a identificação de defeitos ainda não previamente documentados em projetos reais (Nassif, 2023).

A previsão de bugs em software tem se estabelecido como uma estratégia fundamental para garantir a qualidade nos últimos anos, com a capacidade de diminuir gastos e prevenir bugs graves. Entretanto, revisões sistemáticas indicam que os conjuntos de dados comumente utilizados carecem de rótulos e atributos em quantidade suficiente, o que compromete a eficácia das análises. A literatura aponta que muitos conjuntos de dados utilizados na predição de defeitos apresentam problemas como falta de rotulagem adequada, atributos insuficientes e inconsistências que afetam diretamente o desempenho dos modelos. Estudos reforçam que a baixa qualidade dos dados e a validação limitada são fatores críticos para o insucesso de muitos experimentos (Pachouly, 2022).

Muitos estudos têm levado em consideração as técnicas empregadas na predição de defeitos em software com o uso de inteligência artificial, considerando desde a seleção dos conjuntos de dados até as ferramentas empregadas. Diversos trabalhos mostram que os modelos de detecção de defeitos dependem fortemente da qualidade dos conjuntos de dados, que muitas vezes apresentam desbalanceamento, poucos atributos relevantes e validação insuficiente. Essas limitações são discutidas em estudos recentes sobre predição de defeitos em software (Shi, 2023).

A implementação de práticas de qualidade em projetos de software livre apresenta desafios específicos. Isso se deve, principalmente, à diversidade de colaboradores envolvidos com diferentes níveis de experiência e ao caráter descentralizado do desenvolvimento. A utilização de ferramentas adequadas, por sua vez, possibilita o controle da qualidade do projeto, assegurando a adoção de boas práticas na escrita do código (Souza, 2022).

Para tanto, o presente trabalho propõe a realização de uma avaliação comparativa entre os modelos Random Forest e MLP na detecção de bugs, destacando a proposta mais eficaz na detecção de bugs e com melhor usabilidade. O objetivo deste estudo é apoiar desenvolvedores e equipes de Quality Assurance (QA) na escolha de soluções mais adequadas, aumentando a eficiência na detecção precoce de defeitos e reduzindo custos e riscos em produção.

METODOLOGIA

Para a execução do presente trabalho foram adotados elementos metodológicos para comparar os modelos MLP e Random Forest na detecção de falhas em código-fonte. Deste modo, foram selecionadas ferramentas, etapas e procedimentos, visando garantir com clareza, reprodutibilidade e alinhamento aos objetos do estudo.

Este estudo caracteriza-se por ser descritivo, de natureza aplicada, voltado à comparação experimental de dois modelos de aprendizado de máquina para detecção de falhas em código-fonte, com destaque para o MLP e o Random Forest. De acordo com Ramos et al. (2024), pesquisas desse tipo buscam promover melhorias nos processos e produtos de software, por meio da implementação de novas metodologias, ferramentas ou práticas, fundamentadas em análise sistemática e avaliação contínua, sendo amplamente aplicáveis no contexto das Tecnologias da Informação.

O trabalho tem como base a abordagem aplicada, pois busca gerar debates e subsídios para a tomada de decisão sobre qual técnica de aprendizado utilizar em cenários de detecção de bugs, considerando desempenho, precisão e viabilidade prática. A natureza da pesquisa é quantitativa e experimental, visto que serão utilizadas métricas objetivas para avaliar o desempenho dos modelos e experimentos controlados para garantir a reprodutibilidade dos resultados (Gil, 2008). A metodologia para o desenvolvimento do trabalho será dividida em três etapas principais:

1. Coleta e preparação dos dados: Foram utilizados datasets públicos amplamente adotados, como PROMISE (kc1) e CodeXGLUE (function) (Lu et al, 2021), garantindo reprodutibilidade e comparabilidade. Os dados foram preparados e divididos em conjuntos de treino e teste.
2. Implementação e treinamento dos modelos: Os modelos Random Forest e MLP foram implementados em Python com scikit-learn e avaliados por métricas como Accuracy, Precision, Recall e F1-Score.
3. Análise comparativa e validação: Os resultados foram comparados quantitativamente com testes estatísticos e de unidade. Essa combinação permite avaliar tanto o desempenho numérico quanto a relevância prática dos modelos em cenários reais de QA.

O processo segue os princípios da Design Science Research (DSR), conforme Hevner et al. (2004), com ciclos iterativos de desenvolvimento, avaliação e refinamento, assegurando resultados baseados em evidências e aplicáveis à prática.

Fluxo Metodológico

A execução do projeto seguiu uma abordagem estruturada, organizada em etapas sequenciais, conforme ilustrado na Figura 1:

Figura 1: Fluxo metodológico.

Fonte: Elaboração própria (2025).

Inicialmente, definiu-se a temática do trabalho, voltada à comparação de algoritmos supervisionados para detecção de falhas em código-fonte, orientando os experimentos. Foram utilizados os datasets PROMISE (kc1) e CodeXGLUE (function), contendo dados rotulados e não rotulados, preparados e divididos em conjuntos de treino e teste. Os modelos Random Forest e MLP foram implementados com a biblioteca scikit-learn e avaliados pelas métricas Accuracy, Precision, Recall e F1-Score.

Os resultados foram analisados estatisticamente, permitindo comparações quantitativas quanto ao desempenho, uso de dados e capacidade de generalização. Por fim, realizou-se a comparação entre os algoritmos, destacando contribuições, limitações e possibilidades de trabalhos futuros.

Mapeamento Sistemático de Trabalhos e Análise

Foi realizado um mapeamento sistemático da literatura para identificar a aplicação de técnicas de aprendizado de máquina, especialmente Random Forest e MLP, na predição de defeitos em código-fonte.

A busca foi conduzida no Google Acadêmico, utilizando a seguinte string da pesquisa: ("Bug Detection" AND "Source Code" AND "Supervised Learning") OR ("MLP" AND "Random Forest") OR "Software Defect Prediction"

Com o objetivo de garantir um panorama atualizado da área, foram definidos critérios de inclusão publicados a partir de 2019, em inglês ou português, e revisados por pares. O processo de seleção seguiu um funil metodológico estruturado em quatro etapas:

- Identificação inicial (600 resultados): retorno bruto da busca com base na string e no recorte temporal;
- Triagem de relevância (220 trabalhos): exclusão de estudos fora do escopo, como aqueles focados em segurança, criptografia ou sem aplicação em código-fonte;
- Elegibilidade (20 trabalhos): seleção de estudos que aplicam técnicas de aprendizado de máquina na predição de defeitos;

- Seleção final (5 trabalhos): escolha dos estudos mais relevantes para análise comparativa, considerando aderência ao tema, qualidade metodológica e uso de datasets reconhecidos.

Após a seleção final, os trabalhos foram analisados quanto às técnicas utilizadas, tipos de datasets, métricas de avaliação e principais resultados. As informações extraídas foram organizadas e comparadas, permitindo identificar padrões, limitações e lacunas na literatura. Essa análise serviu de base para a construção da abordagem proposta neste estudo.

Ferramentas e Estrutura Utilizada

Os experimentos foram conduzidos em ambiente que garante reprodutibilidade e comparabilidade, utilizando Python com as bibliotecas scikit-learn, pandas, numpy e matplotlib, além do Google Colab como plataforma de desenvolvimento. Foram empregados os datasets públicos PROMISE (kc1) e CodeXGLUE (function), amplamente utilizados na literatura.

A estrutura experimental envolveu:

1. preparação dos dados com divisão em treino e teste;
2. treinamento e avaliação dos modelos Random Forest e MLP com métricas como Accuracy, Precision, Recall e F1-Score;
3. análise comparativa com apoio de métodos estatísticos e visualizações.

RESULTADOS

Este capítulo apresenta os resultados obtidos com o desenvolvimento do estudo, abordando a estrutura metodológica e seus requisitos técnicos. Os resultados esperados visam comprovar a eficiência comparativa entre os algoritmos Random Forest (aprendizado supervisionado) e MLP (aprendizado supervisionado) na detecção de falhas em código-fonte.

Resultados do Mapeamento Sistemático

A partir do mapeamento sistemático da literatura, foram selecionados estudos representativos sobre predição de defeitos em software conforme apresentado na Tabela 1.

Tabela 1: Resultados do Mapeamento Sistemático.

Autor	Ano	Técnica	Contribuição
Nassif	2023	MLP	Alto desempenho em padrões complexos
Pachouly	2022	Revisão	Problemas com datasets

Shi	2023	RF / ML	Impacto do desbalanceamento
Daza	2025	IA	Aplicações industriais

Fonte: Elaboração própria (2026).

O trabalho de Nassif (2023) utiliza redes neurais para identificar padrões complexos em dados de software. Como principal valência, destaca-se a capacidade de modelar relações não lineares, resultando em bom desempenho em tarefas de classificação. Como limitação, apresenta dependência de grandes volumes de dados, maior custo computacional e baixa interpretabilidade.

O estudo de Shi (2023) analisa o impacto do desbalanceamento de dados na predição de defeitos. Como contribuição, propõe melhorias no tratamento desses dados, aumentando a eficácia dos modelos. Entretanto, o desempenho ainda depende fortemente da qualidade dos dados e da seleção adequada de atributos.

Já Pachouly (2022) apresenta uma revisão sistemática sobre o uso de inteligência artificial na área. Como valência, oferece uma visão consolidada sobre datasets, técnicas e desafios. Como limitação, evidencia problemas recorrentes, como baixa qualidade dos dados e falta de padronização experimental.

Comparação com a Literatura e Lacuna de Pesquisa

A análise dos trabalhos selecionados evidencia lacunas importantes na literatura de predição de defeitos em software. Observa-se que muitos estudos avaliam algoritmos de forma isolada, sem realizar comparações diretas sob as mesmas condições experimentais. Além disso, há forte dependência da qualidade dos datasets, bem como desafios relacionados ao desbalanceamento de dados e à generalização dos modelos.

Diferentemente das abordagens existentes, o presente trabalho propõe uma avaliação comparativa entre Random Forest e MLP utilizando os mesmos *datasets*, métricas e testes estatísticos, permitindo uma análise mais controlada e justa do desempenho dos algoritmos.

Enquanto estudos como Nassif (2023) focam no potencial das redes neurais e Shi (2023) abordam problemas específicos como desbalanceamento, este trabalho busca integrar essas perspectivas por meio de uma comparação prática e experimental.

Dessa forma, a principal contribuição desta pesquisa está na análise comparativa entre modelos amplamente utilizados, considerando diferentes tipos de dados (estruturados e textuais), o que ainda é pouco explorado na literatura.

Descrição dos Experimentos

Optou-se pela utilização do dataset KC1, do repositório PROMISE, por ser amplamente adotado na literatura de predição de defeitos, permitindo comparabilidade entre estudos e avaliação dos modelos em dados estruturados. Além disso, foi utilizado o dataset function.json, estruturado em nível de função, possibilitando a análise de código-fonte textual e a comparação do desempenho dos modelos em diferentes representações de dados.

Para o dataset KC1, proveniente do PROMISE Repository, os dados foram inicialmente organizados em variáveis de entrada (X) e saída (y), onde X representa as métricas de código e y a classificação de defeito (presença ou ausência de bugs).

A divisão entre conjunto de treinamento e teste foi realizada utilizando a função train_test_split, adotando-se 70% dos dados para treinamento e 30% para teste (test_size=0.3), com random_state=42, garantindo a reprodutibilidade dos experimentos.

Após a divisão, foi treinado o modelo *Random Forest*, configurado com 100 árvores de decisão (n_estimators=100) e random_state=42. O treinamento foi realizado por meio do método fit, utilizando apenas os dados de treinamento (X_train e y_train), permitindo que o modelo aprendesse padrões associados à presença de defeitos no código.

Para o dataset function.json, o processo de divisão dos dados seguiu a mesma abordagem utilizada no dataset KC1, garantindo consistência metodológica entre os experimentos. Os dados foram divididos em 70% para treinamento e 30% para teste, utilizando a função train_test_split com random_state=42.

Entretanto, diferentemente do dataset KC1, que já possui atributos numéricos estruturados, o dataset function.json contém código-fonte em formato textual. Dessa forma, foi necessária uma etapa adicional de pré-processamento, denominada vetorização. Para isso, foi utilizado o método CountVectorizer, limitando o vocabulário a 5000 atributos (max_features=5000). Esse processo transforma o código-fonte em representações numéricas baseadas na frequência de termos, permitindo sua utilização em modelos de aprendizado de máquina.

O vetor de treinamento foi ajustado com fit_transform sobre os dados de treino (X_train), enquanto os dados de teste (X_test) foram transformados utilizando apenas transform, evitando vazamento de dados (data leakage).

Os experimentos foram implementados em Python utilizando a biblioteca scikit-learn, seguindo um fluxo estruturado e padronizado. Inicialmente, os dados

foram organizados em variáveis de entrada (X) e saída (y), sendo posteriormente divididos em conjuntos de treino e teste (70%/30%) com uso da função train_test_split e random_state=42. Para o dataset KC1, foram utilizadas diretamente métricas estruturadas, enquanto no dataset function.json o código-fonte foi transformado em representações numéricas por meio do CountVectorizer. Os modelos Random Forest e MLP foram então treinados separadamente utilizando os mesmos dados de treinamento, e avaliados com base nas métricas de acurácia, precisão, recall e F1-score.

Configuração do Random Forest

O modelo Random Forest foi implementado utilizando a classe RandomForestClassifier da biblioteca Scikit-learn. Para ambos os datasets, KC1 dataset e function.json dataset, foi adotada uma configuração padrão com 100 árvores de decisão (n_estimators=100) e random_state=42, garantindo a reprodutibilidade dos experimentos.

O treinamento foi realizado por meio do método fit, utilizando os dados de treinamento previamente separados. Após o treinamento, o modelo foi aplicado ao conjunto de teste por meio do método predict, gerando as previsões utilizadas na avaliação.

Para medir o desempenho do modelo, foram utilizadas as métricas de accuracy, precision, recall e F1-score, amplamente adotadas em problemas de classificação binária. No caso do dataset KC1, foi necessário especificar o parâmetro pos_label='_TRUE', devido à forma como a classe positiva está representada no dataset.

Além disso, foi utilizada a matriz de confusão para análise detalhada dos resultados, permitindo observar a distribuição entre verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos.

Configuração do MLP

O modelo de rede neural foi implementado utilizando a classe MLPClassifier, representando um Perceptron Multicamadas (MLP), um algoritmo de aprendizado supervisionado amplamente utilizado em tarefas de classificação. Para ambos os datasets, foi utilizada uma arquitetura composta por uma única camada oculta com 100 neurônios (hidden_layer_sizes=(100,)), permitindo ao modelo capturar padrões não lineares nos dados.

No dataset KC1, o modelo foi configurado com um limite máximo de 500 iterações (max_iter=500), enquanto no dataset function.json foi utilizado um limite de 300 iterações (max_iter=300). Essa diferença se deve à natureza dos dados, uma vez que o dataset vetorizado tende a demandar maior custo computacional.

Assim como no Random Forest, foi utilizado random_state=42 para garantir reprodutibilidade. Após o treinamento com o método fit, o modelo gerou previsões por meio do método predict, sendo avaliado com as mesmas métricas: *accuracy*, *precision*, *recall* e F1-score. Para o dataset KC1, também foi necessário definir explicitamente o rótulo positivo (pos_label='_TRUE').

Metricas da Avaliação

Para a realização dos experimentos, foi desenvolvido um código em Python utilizando a biblioteca scikit-learn, responsável pelo treinamento, teste e avaliação dos modelos Random Forest e MLP.

No dataset KC1 (PROMISE), os dados foram carregados a partir de um arquivo no formato ARFF, contendo métricas estruturadas de código, como complexidade, linhas de código e acoplamento. A variável alvo utilizada foi “DL”, que indica a presença (_TRUE) ou ausência (_FALSE) de defeitos no módulo. Os dados foram divididos em conjuntos de treino e teste (70%/30%) e utilizados diretamente nos modelos, sem necessidade de vetorização.

Já no dataset function.json, os dados consistem em código-fonte textual no nível de função, onde a variável “func” representa o código e “target” indica a presença (1) ou ausência (0) de defeitos. Para viabilizar o uso nos modelos, foi aplicada vetorização textual por meio do método CountVectorizer, transformando o código em representações numéricas de alta dimensionalidade.

Os modelos Random Forest (100 árvores) e MLP (rede neural com uma camada oculta de 100 neurônios) foram treinados utilizando os mesmos conjuntos de dados, garantindo comparabilidade. A divisão treino/teste foi mantida consistente, com random_state fixo para reprodutibilidade.

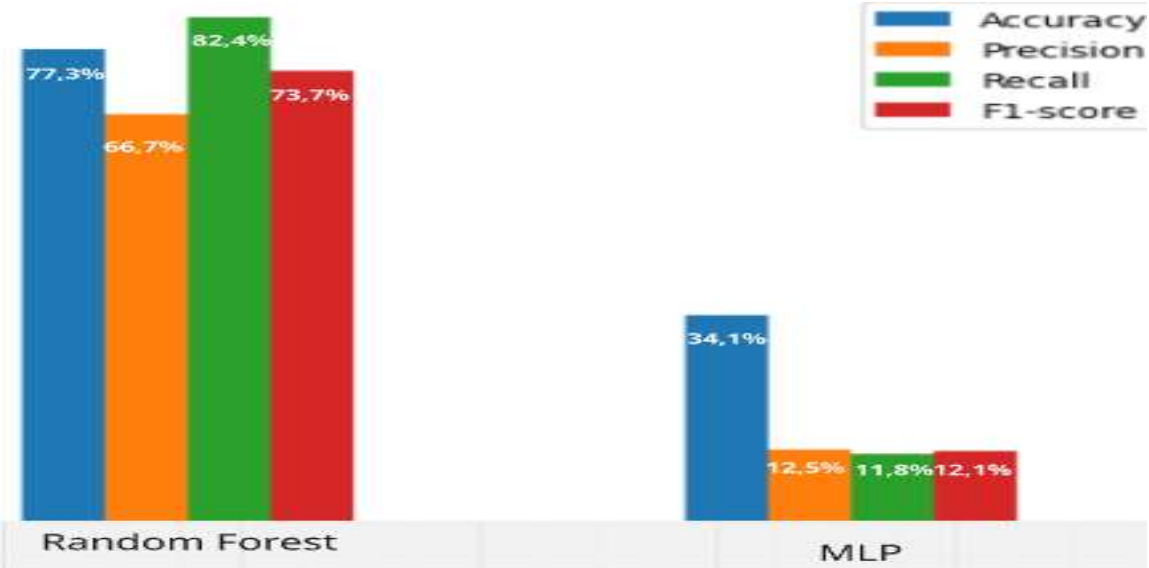
Os defeitos considerados nos datasets correspondem à classificação binária de código defeituoso ou não defeituoso, baseada em dados históricos. O objetivo dos modelos foi identificar automaticamente padrões associados à presença de falhas no código.

Por fim, o desempenho dos modelos foi avaliado por meio das métricas acurácia, precisão, recall e F1-score, sendo esta última adotada como principal devido ao desbalanceamento dos dados.

Resultados no Dataset KC1 (PROMISE)

Os experimentos realizados no dataset KC1 dataset tiveram como objetivo comparar o desempenho dos modelos Random Forest e MLP na tarefa de detecção de defeitos em código. Os resultados obtidos demonstram que o modelo Random Forest apresentou desempenho significativamente superior ao modelo MLP em todas as métricas avaliadas conforme ilustrado na Figura 2.

Figura 2: Gráfico de Comparação dos Modelos no promise.



Fonte: Elaboração própria (2026).

Em termos de acurácia, o Random Forest atingiu aproximadamente 77,3%, enquanto o MLP obteve apenas 34,1%, evidenciando uma grande diferença na capacidade de classificação geral.

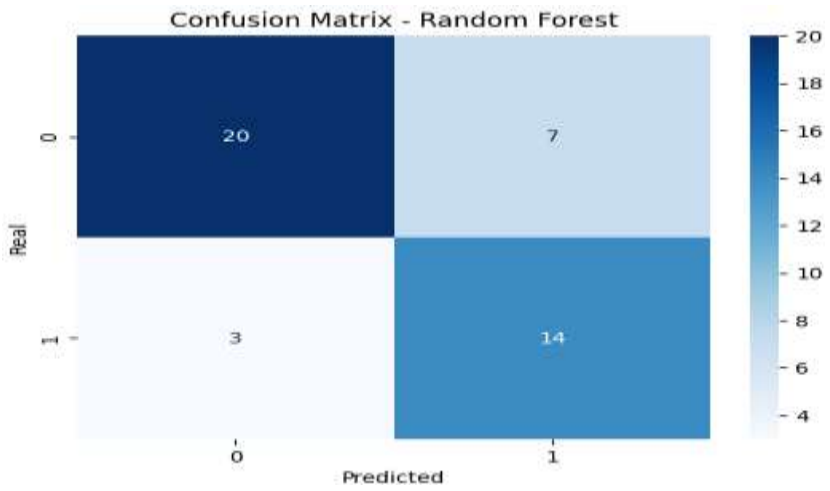
Em relação ao F1-score, métrica considerada mais relevante neste trabalho devido ao desbalanceamento dos dados, o Random Forest alcançou 0,737, enquanto o MLP apresentou desempenho bastante inferior, com apenas 0,121.

Além disso, o Random Forest demonstrou maior capacidade de detecção de defeitos, conforme indicado pelo recall (0,824), indicando que o modelo conseguiu identificar a maioria dos bugs presentes no dataset.

Por outro lado, o modelo MLP apresentou baixo desempenho em todas as métricas, sugerindo dificuldade em aprender padrões relevantes a partir dos dados disponíveis.

Esses resultados indicam que modelos baseados em árvores, como o Random Forest, são mais adequados para dados estruturados, como os presentes no dataset KC1, enquanto o MLP pode exigir ajustes adicionais ou maior volume de dados para apresentar desempenho competitivo conforme apresentado na Figura 3.

Figura 3: Matriz de Confusão (Random Forest).



Fonte: Elaboração própria (2026).

Figura 4: Análise Estática (Pylint) no promise.

```
!pylint modelo_tcc.py

************* Module modelo_tcc
modelo_tcc.py:1:0: C0114: Missing module docstring (missing-module-
modelo_tcc.py:6:0: C0116: Missing function or method docstring (mis
modelo_tcc.py:6:26: C0103: Argument name "X_train" doesn't conform
modelo_tcc.py:14:0: C0116: Missing function or method docstring (mi
modelo_tcc.py:14:0: C0103: Function name "treinar_MLP" doesn't conf
modelo_tcc.py:14:16: C0103: Argument name "X_train" doesn't conform
modelo_tcc.py:2:0: W0611: Unused pandas imported as pd (unused-impo

-----------------------------------
Your code has been rated at 3.64/10
```

Fonte: Elaboração própria (2026).

A análise estática do código identificou alguns pontos de melhoria, como ausência de docstrings, convenções de nomenclatura e imports não utilizados. Apesar disso, não foram encontrados erros críticos que comprometessem a execução dos experimentos conforme ilustrado na Figura 4.

Figura 5: Testes de Unidade (Pytest) no promise.

```
!pytest -v

============================= test session starts ==============
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /us
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.4.5, typeguard-4.4.4
collected 2 items

test_modelo_tcc.py::test_random_forest_treinamento PASSED
test_modelo_tcc.py::test_MLP_treinamento PASSED

============================== 2 passed in 1.68s ===============
```

Fonte: Elaboração própria (2026).

Os testes de unidade foram realizados com o framework pytest, com o objetivo de validar o correto funcionamento das funções de pré-processamento, divisão dos dados e treinamento dos modelos. Foram verificados aspectos como carregamento dos datasets, consistência das dimensões após a divisão treino/teste e execução dos

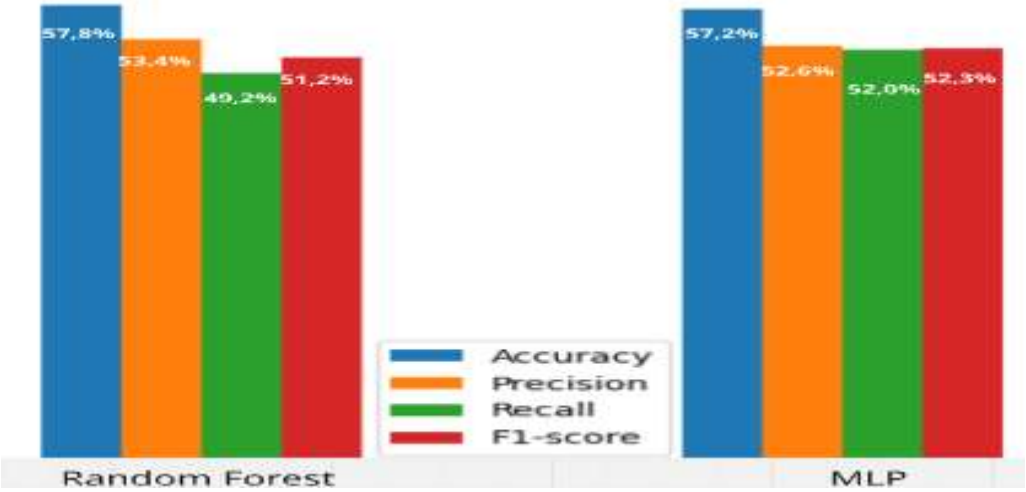
algoritmos sem erros. Esses testes contribuíram para garantir a confiabilidade e a reprodutibilidade do experimento conforme apresentado na Figura 5.

Resultados no Dataset function.json

Os experimentos no dataset function.json avaliaram o desempenho dos modelos Random Forest e MLP em dados de código-fonte vetorizados. Os resultados obtidos demonstram que, diferentemente do observado no dataset KC1, o desempenho dos modelos Random Forest e MLP foi bastante próximo no dataset function.json.

Em termos de acurácia, ambos os modelos apresentaram valores semelhantes, com leve vantagem para o Random Forest (0,578) em relação ao MLP (0,573). No entanto, ao analisar o F1-score, métrica mais relevante para este trabalho, o modelo MLP apresentou desempenho ligeiramente superior (0,524) em comparação ao Random Forest (0,513), indicando melhor equilíbrio entre precisão e recall conforme apresentado na Figura 6.

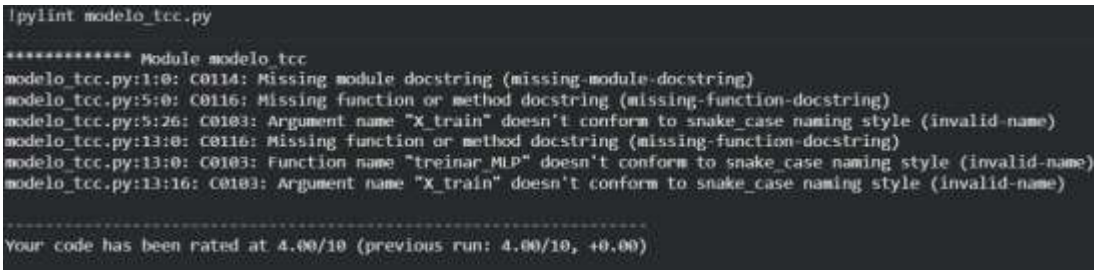
Figura 6: Gráfico de Comparação dos Modelos no function.



Fonte: Elaboração própria (2026).

Além disso, o MLP demonstrou maior capacidade de generalização em dados textuais vetorizados, possivelmente devido à sua habilidade em capturar padrões mais complexos em representações de alta dimensionalidade. O aviso de convergência no MLP indica necessidade de mais iterações ou ajuste de hiperparâmetros. Em geral, redes neurais apresentaram desempenho competitivo ou superior em dados textuais.

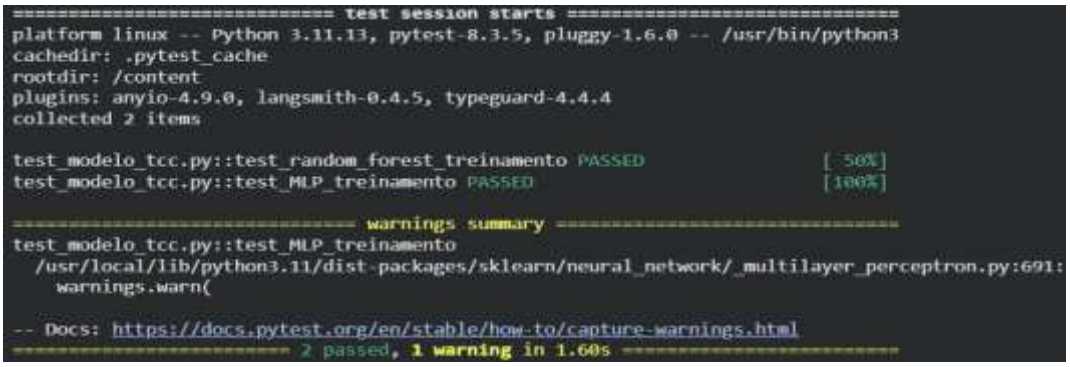
Figura 7: Análise Estática (Pylint) no function.



Fonte: Elaboração própria (2026)

A análise estática identificou problemas relacionados à ausência de docstrings e ao não cumprimento de convenções de nomenclatura (snake_case). Apesar disso, não foram encontrados erros críticos que comprometessem a execução dos modelos conforme apresentado na Figura 7.

Figura 8: Testes de Unidade (Pytest) no function.



Fonte: Elaboração própria (2026)

A Figura 8 apresenta o resultado dos testes de unidade realizados com pytest, nos quais foram executados dois testes referentes ao treinamento dos modelos Random Forest e MLP, ambos concluídos com sucesso (“PASSED”), indicando o funcionamento correto das funções implementadas. Observa-se também um aviso no MLP relacionado à não convergência dentro do limite de 300 iterações, o que sugere a necessidade de ajuste de hiperparâmetros. De forma geral, os resultados confirmam a correta execução dos testes e a confiabilidade do experimento.

Comparação Geral dos Modelos

A comparação entre Random Forest e MLP considerou os resultados nos datasets KC1 e function.json, permitindo avaliar o comportamento dos modelos em diferentes tipos de dados. No KC1, composto por métricas estruturadas, o Random Forest apresentou desempenho superior em todas as métricas, com destaque para o F1-score de 0,737, frente a 0,121 do MLP, indicando maior capacidade de detecção de defeitos.

No function.json, baseado em código-fonte textual vetorizado, os resultados foram mais equilibrados. O MLP obteve F1-score de 0,524, enquanto o Random Forest alcançou 0,513, apresentando leve vantagem para a rede neural. Essa diferença está relacionada à natureza dos dados: o Random Forest é mais eficaz em dados estruturados e de menor dimensionalidade, como no KC1, enquanto o MLP modela melhor relações complexas em dados de alta dimensionalidade, como no function.json.

Em termos de consistência, o Random Forest foi superior no KC1, mas perdeu desempenho relativo no dataset textual. O MLP apresentou baixo desempenho no KC1, porém foi mais competitivo no function.json, indicando maior adaptabilidade conforme apresentado na Tabela 2.

Tabela 2: Comparação de desempenho entre Random Forest e MLP.

Dataset	Modelo	F1-score	Desempenho Geral
KC1	Random Forest	0,737	Superior
KC1	MLP	0,121	Inferior
function.json	Random Forest	0,513	Competitivo
function.json	MLP	0,524	Superior

Fonte: Elaboração própria (2026).

De forma geral, não há modelo universalmente superior; a escolha depende do tipo de dado. Neste estudo, o Random Forest foi mais adequado para dados estruturados, enquanto o MLP teve melhor desempenho em dados textuais vetorizados.

Análise Estatística

Para verificar a significância das diferenças entre os modelos Random Forest e MLP, foi aplicado o teste não paramétrico de Wilcoxon, adequado para dados sem suposição de normalidade, permitindo avaliar se as diferenças observadas nas métricas são estatisticamente relevantes ou resultam de variações aleatórias, aumentando a confiabilidade das conclusões do estudo.

O teste de Wilcoxon é um método não paramétrico utilizado para comparar duas amostras relacionadas, sem exigir normalidade dos dados. Ele avalia se as diferenças entre os modelos são significativas ou aleatórias. Neste estudo, foi aplicado às métricas dos modelos Random Forest e MLP.

Resultados — Dataset KC1

Para o dataset KC1, o teste de Wilcoxon resultou em um p-valor igual a 1.0, indicando que não há evidência estatística suficiente para afirmar que existe diferença significativa entre os modelos.

Esse resultado pode estar relacionado ao número limitado de observações ou à baixa variabilidade dos dados. Também foi observado um aviso durante o teste, indicando possíveis limitações do método nesse cenário.

Assim, embora o Random Forest tenha apresentado melhor desempenho nas métricas, não é possível afirmar que a diferença seja estatisticamente significativa.

Resultados — Dataset function.json

Para o dataset function.json, o teste de Wilcoxon resultou em estatística igual a 0.0 e p-valor de 0.001953125, indicando que há evidência estatística suficiente para afirmar a existência de diferença significativa entre os modelos.

Esse resultado apresentou p-valor inferior a 0.05, indicando diferença estatisticamente significativa.

Esse resultado demonstra que a diferença observada no desempenho não ocorreu ao acaso, sendo estatisticamente relevante.

Assim, pode-se afirmar que, para esse dataset, o modelo MLP apresentou desempenho significativamente diferente e ligeiramente superior em relação ao Random Forest.

Interpretação Geral

A análise estatística indicou comportamentos distintos: no KC1, não houve significância apesar das diferenças nas métricas, enquanto no function.json a diferença foi estatisticamente significativa. Esses resultados reforçam a importância da análise estatística para evitar conclusões baseadas apenas em métricas descritivas.

CONSIDERAÇÕES FINAIS

Os resultados mostram que o desempenho de Random Forest e MLP varia conforme a natureza dos dados. No dataset KC1, o Random Forest foi significativamente superior devido ao caráter estruturado dos dados, que favorece modelos baseados em árvores capazes de lidar com atributos numéricos e relações não lineares.

O MLP apresentou baixo desempenho nesse cenário, possivelmente pela menor capacidade de generalização em conjuntos menores e pela necessidade de maior volume de dados e ajuste de hiperparâmetros.

No dataset function.json, os modelos tiveram desempenho mais equilibrado, com leve vantagem do MLP em F1-score, devido à representação vetorial de alta dimensionalidade, na qual redes neurais capturam melhor padrões complexos.

Não há modelo universalmente superior; a escolha depende de características como desbalanceamento, ruído e representação dos dados. A análise estatística também mostrou que nem todas as diferenças são significativas, reforçando a necessidade de validação rigorosa.

REFERÊNCIAS

DAZA, A. Industrial applications of artificial intelligence in software defects prediction: systematic review, challenges, and future works. **Computers and Electrical Engineering**, v. 121, 2025. Disponível em: https://www.sciencedirect.com. Acesso em: 30 mar. 2026.

GIL, A. C. **Métodos e técnicas de pesquisa social**. 6. ed. São Paulo: Atlas, 2008. Disponível em: https://feata.edu.br/downloads/revistas/economiaepesquisa/v3_artigo01_globalizacao.pdf. Acesso em: 03 abr. 2026.

HEVNER, A. R. Design science in information systems research. **MIS Quarterly**, v. 28, n. 1, p. 75–105, 2004. Disponível em: https://www.in.th-nuernberg.de/professors/Holl/Personal/Hevner_DesignScience_ISRes.pdf. Acesso em: 03 abr. 2026.

LU, S. et al. CodeXGLUE: **A machine learning benchmark dataset for code understanding and generation**. 2021. Disponível em: https://github.com/microsoft/CodeXGLUE. Acesso em: 30 mar. 2026.

NASSIF, A. B. Software defect prediction using learning to rank approach. **Scientific Reports**, v. 13, 2023. Disponível em: https://www.nature.com/articles/s41598-023-45915-5.pdf. Acesso em: 30 mar. 2026.

PACHOULY, J. A systematic literature review on software defect prediction using artificial intelligence: datasets, data validation methods, approaches, and tools. **Engineering Applications of Artificial Intelligence**, v. 111, 2022. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0952197622000616. Acesso em: 30 mar. 2026.

PROMISE REPOSITORY. **Software engineering data repository**. Disponível em: http://promise.site.uottawa.ca/SERepository/. Acesso em: 30 mar. 2026.

RAMOS, Rommel Gabriel Gonçalves; TEIXEIRA, Jhonatas Moreira; NETTO, Itamar da Silva Bonfim; SOUZA, Anderson Ferreira de; PIANTINO, Luiz Fernando Moura. Qualidade de software: análise de dados e proposta de melhoria de processo.

Revista Sociedade Científica, 2024. Disponível em: https://journal.scientificsociety.net. Acesso em: 22 abr. 2026.

SHI, H. Improving software defect prediction in noisy imbalanced datasets. **Applied Sciences**, v. 13, n. 18, 2023. Disponível em: https://www.mdpi.com/2076-3417/13/18/10466. Acesso em: 30 mar. 2026.

SOUZA, C. E. **Análise de qualidade de código em projetos de software livre:** um estudo de caso no GNU Health. São Paulo, 2022. Dissertação (Mestrado em Ciências da Computação) – Universidade de São Paulo. Disponível em: https://repositorio.usp.br/item/003200204. Acesso em: 30 mar. 2026.