Aprendizado de Máquina e Métricas Halstead na Predição de Falhas de *Software*

José Marcos Gomes[1], **Prof.**[a] **Dr.**[a] **Ana Carolina Lorena**[1], **Prof. Dr. Luiz Alberto Vieira Dias**[1], **Prof. Dr. Denis Loubach**[1]

[1]Instituto Tecnológico de Aeronáutica - ITA
Divisão da Ciência da Computação - IEC
São José dos Campos/SP - Brasil

gomesjm@ita.br, aclorena@gmail.com, vdias@ita.br, dloubach@ita.br

Abstract. *The requirements and efforts necessary to guarantee the quality of the software produced by the industry are very large and difficult to overcome. Despite significant advances in the last decades, the task is far from considered fast and easy.*
The artificial intelligence area has had a revival with the adoption of machine learning techniques and the advent of neural networks. The application of these techniques in quality control processes has been very large in several sectors of the economy.
Three experiments conducted with publicly and freely available data sets were carried out and encouraging results were obtained. Although the hit rate is not high, it is possible to use the technique effectively in prioritizing efforts to determine programs that are more prone to failure.
There are several studies using machine learning techniques to assess software quality, however the low accuracy rate of this method can discourage practitioners from adopting the technique. We provide information on the pros and cons of this approach and make recommendations for industry practitioners.

Resumo. *Os custos e esforços necessários para se garantir a qualidade do software produzido pela indústria são muito grandes e difíceis de sobrepor. Apesar dos significativos avanços conseguidos nas últimas décadas, a tarefa está longe de ser considerada fácil e rápida.*
A área de inteligência artificial tem tido um renascimento com a adoção das técnicas de aprendizado de máquina e o advento das redes neurais. A aplicação destas técnicas em processos de controle de qualidade tem sido muito grande em diversos setores da economia.
Três experimentos conduzidos com conjuntos de dados disponíveis gratuitamente e publicamente foram feitos e resultados animadores foram obtidos. Apesar da taxa de acertos não ser alta, é possível utilizar a técnica de forma efetiva na priorização dos esforços em determinar programas mais sujeitos à falhas.
Existem diversos estudos utilizando técnicas de aprendizado de máquina na aferição da qualidade de software, porém o baixo índice de acurácia deste método pode desanimar os praticantes a adotarem a técnica. Foram avaliados os prós e contras desta abordagem e fizemos recomendações gerais para os praticantes da indústria.

1. Introdução

1.1. Problema

Nos Estados Unidos, em 2002 o NIST estimou um custo de 59.5 bilhões de dólares decorrente de infraestrutura de testes inadequada [Planning 2002].

	Testadores / Funcionários (milhões)	Custo da infraestrutura de testes inadequada		Redução de custo potencial com melhorias viáveis.	
		Custo por	Custo total (milhões US$)	Custo por	Custo total (milhões US$)
Desenvolvedores	**0,302**	**69.945**	**21.155**	**34.964**	**10.575**
Usuários					
Indústria	**25,0**	**459**	**11.463**	**135**	**3.375**
Serviços	**74,1**	**362**	**26.858**	**112**	**8.299**
Total			**59.477**		**22.249**

Tabela 1. Estimativa de impacto nacional nos EUA (adaptado de [Planning 2002])

Segundo o relatório do NIST, a estimativa de impacto nacional de uma infraestrutura inadequada para teste de "*software*" é de 859 bilhões e a redução de custo potencial de melhorias viáveis é de 822 bilhões. Os usuários de "*software*" respondem por uma parcela maior dos custos totais de infraestrutura inadequada (64 por cento) em comparação com as reduções de custos viáveis (52 por cento) porque uma grande parte dos custos dos usuários se deve a atividades de prevenção. Considerando que as atividades de mitigação diminuem proporcionalmente com a diminuição na incidência de *bugs* e erros, os custos de prevenção (como sistemas redundantes e investigação de decisões de compra) provavelmente persistirão, mesmo que apenas alguns erros sejam esperados. Para desenvolvedores de "*software*", a economia de custo viável é de aproximadamente 50 por cento dos custos totais de infraestrutura inadequada. Isso reflete uma diminuição mais proporcional no esforço de teste à medida que os recursos e ferramentas de teste melhoram.

1.2. Contribuição

Umas das formas de se reduzir o custo de uma operação é por meio da automação, o que pode ser observado desde a primeira revolução industrial e a substituição do homem pela máquina na linha de produção de bens que a cada dia se tornam mais acessíveis. Uma das razões para se adotar automação dos testes de "*software*" inclue, por exemplo, o tempo tomado por testes manuais, que a automação irá aumentar a eficiência, especialmente em razão dos testes de regressão, onde os testes são executados após modificações no "*software*" [Varma 2000]. Quando aplicável, a automação do teste de "*software*" pode reduzir custos, que podem representar até 50 por cento do custo total do desenvolvimento de "*software*" [Kit and Finzi 1995].

Outra forma de se reduzir custos é a otimização, onde se busca consumir a menor quantidade de recursos na produção da mesma quantidade de produtos.

Neste trabalho pretendemos investigar o uso decisões técnicas de inteligência artificial na detecção de problemas potenciais em "*software*" e deste modo automatizar e otimizar o trabalho dos desenvolvedores.

2. Metodologia

2.1. Métricas de Qualidade de Software

As métricas de Halstead [Halstead et al. 1977] se dividem em dois grupos:

1. Medições básicas:
    - $uniq\_Op$: *unique operators* - operadores únicos
    - $uniq\_Opnd$: *unique operands* - operandos únicos
    - $total\_Op$: *total operators* - total de operadores
    - $total\_Opnd$: *total operands* - total de operandos
    - n: *line number* - número de linhas
2. Medições derivadas:
    - v: Halstead *volume* - volume
    - l: Halstead *program length* - comprimento do programa
    - d: Halstead *difficulty* - dificuldade
    - i: Halstead *intelligence* - inteligência
    - e: Halstead *effort* - esforço
    - b: Halstead *numeric* - número
    - t: Halstead *time estimator* - estimativa de tempo

onde:

$$
\begin{aligned}
v =& (2 + uniq\_Opnd) \times \log_2(2 + uniq\_Opnd) \\
l =& uniq\_Opnd \log_2 uniq\_Opnd \times uniq\_Op \log_2 uniq\_Op \\
d =& \frac{uniq\_Op}{2} \times \frac{total\_Opnd}{uniq\_Opnd} \\
i =& \frac{v}{d} \\
e =& d \times v \\
t =& \frac{e}{18} \ seconds
\end{aligned}
$$

2.2. Análise dos Resultados

Introduzida pelo bioquímico Brian W. Matthews em 1975., o MCC[i], ou coeficiente ϕ, é usado em ML como medição da qualidade das classificações binárias (duas classes).

O MCC é essencialmente um coeficiente de correlação entre as classificações binárias observadas e previstas; ele retorna um valor entre -1 e $+1$. Um coeficiente de $+1$ representa uma predição perfeita, 0 não melhor do que a predição aleatória e -1 indica discordância total entre predição e observação. O MCC está intimamente relacionado à estatística qui-quadrado para uma tabela de contingência 2×2, e leva em consideração verdadeiros e falsos positivos e negativos, geralmente considerado uma medição balanceada e que pode ser usada mesmo se as classes forem de tamanhos muito diferentes [Boughorbel et al. 2017].

Nossa classificação será determinar se cada instância contendo as medições coletadas está dentro ou não do espectro de casos que possui chance de conter falhas, e com isso priorizar os módulos que merecem maior atenção durante a fase de verificação e validação do desenvolvimento de "*software*".

[i] A análise de MCC foi feita pela biblioteca Python de código aberto PyCM[Haghighi et al. 2018].

$$|MCC| = \sqrt{\frac{x^2}{n}}$$

onde n: número total de observações, ou a partir da matriz de confusão:

$$MCC = \frac{VP \times VN - FP \times FN}{\sqrt{(VP + FP)(VP + FN)(VN + FP)(VN + FN)}}$$

onde VP é o número de **verdadeiros positivos**, VN o número de **verdadeiros negativos**, FP o de **falsos positivos** e FN o de **falsos negativos**. Se qualquer uma das somas resultar em zero, ao denominador pode ser arbitrariamente atribuído o valor um, o que resultará num coeficiente de Mattews igual à zero.

3. Conjuntos de Dados

Os dados e as informações neles contidas e utilizados neste estudo estão publicamente disponíveis em PROMISE "*Software Engineering Repository*" [Sayyad Shirabad and Menzies 2005] e consiste de 21 medições em diversos módulos de programas utilizados em espaçonaves da NASA.

1. loc: McCabe - *line count* - contagem de linhas de código
2. $v(g)$: McCabe - "complexidade ciclomática" - obtida da contagem de vértices v do grafo g do programa
3. $ev(g)$: McCabe - "complexidade essencial" - obtida da medida do grau que um módulo possui de construções não estruturadas[ii] - e representa a qualidade do código
4. $iv(g)$: McCabe "complexidade de desenho" - reflete a complexidade de padrões de chamadas de um módulo, o que diferencia módulos que complicam o desenho da aplicação de módulos que possuem lógica computacional complexa
5. n: Halstead total de operadores e operandos
6. v: Halstead volume
7. l: Halstead comprimento do programa
8. d: Halstead dificuldade
9. i: Halstead inteligência
10. e: Halstead esforço
11. b: Halstead complexidade
12. t: Halstead estimativa de tempo
13. $lOCode$: Halstead *line count* - contagem de linhas de código
14. $lOComment$: Halstead *count of lines of comments* - contagem de linhas de comentários
15. $lOBlank$: Halstead *count of blank lines* - contagem de linhas em branco
16. $lOCodeAndComment$: *line count of code and comments* - contagem de linhas de código e comentários
17. $uniq\_Op$: operadores únicos

[ii] Construções não estruturadas se referem à um grafo cujos nós não podem ser separados de uma unidade de execução ou ao "*loop*" do qual fazem parte, ou que transferem o controle para fora da unidade de execução sem provisão de um retorno ("*go to*")

18. $uniq\_Opnd$: operandos únicos
19. $total\_Op$: total de operadores
20. $total\_Opnd$: total de operandos
21. $branchCount$: *of the flow graph* - fluxos de execução

Além destas, as fontes de dados contam com as seguintes colunas:

- *errorRate* - quantidade de erros
- *defects* - possui erros ($0 = \textit{não}, 1 = \textit{sim}$)

Especificamente, as seguintes fontes de dados de previsão de defeitos foram utilizadas neste trabalho:

- **JM1** - 10885 instâncias
- **CM1** - 497 instâncias
- **KC2** - 532 instâncias

Optamos por utilizar estes arquivos em particular pois todos possuem os mesmos atributos, variando apenas o número de instâncias de cada. Existem ainda diversos outros arquivos semelhantes, que porém exigiriam estudos e customizações particulares para cada caso, o que procuramos evitar neste trabalho.

3.1. Qualidade dos Conjuntos de Dados

Muitos pesquisadores utilizam medições estáticas como guias para a previsão da qualidade do código.

Ainda assim estes métodos são criticados. Fentom [Fenton and Pfleeger 1997] por exemplo, mostra que a mesma funcionalidade implementada em diferentes linguagens de programação resulta em diferentes medições.

Apoiamos o ponto de vista de que estas medições são nada mais do que indicadores probabilísticos de defeitos a serem utilizadas na tomada de decisões e priorização de tarefas.

Apesar das críticas com relação à estes conjuntos de dados, eles são utilizados para treinamento das redes neurais pela maioria dos estudos [Gray et al. 2011, Shepperd et al.] atuais, e dada a natureza desbalanceada dos dados extraídos de qualquer "*software*" [Ryu and Baik 2016] adotamos estes conjuntos e aplicamos sobre eles critérios de pré-processamento listados a seguir.

3.2. Pre-processamento de Dados

Fatores apontados por Gray [Gray et al. 2011] que listamos a seguir foram considerados neste trabalho:

- Eliminação de todas as colunas exceto as pertinentes a cada classe de medição adotada no treinamento (Halstead ou McCabe)
- Remover linhas com valores nulos ou em branco
- Remover linhas com valores não numéricos
- Normalização dos dados
- Discretização dos dados utilizados na classificação
- Balanceamento dos dados

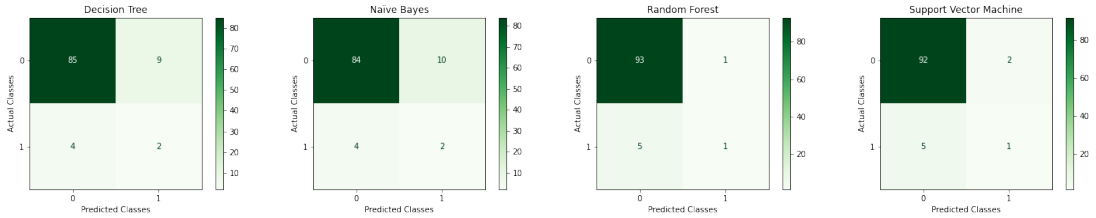

Figura 1. Matriz de Confusão - Conjunto de dados CM1

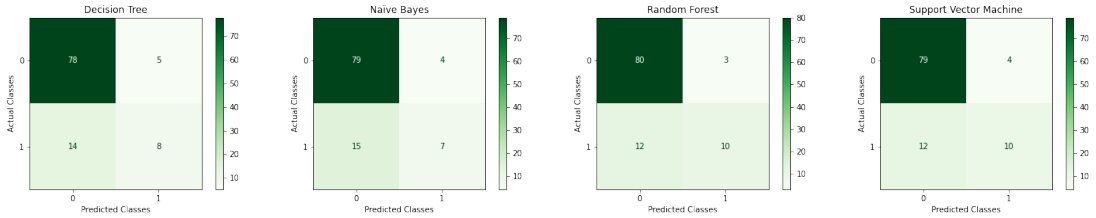

Figura 2. Matriz de Confusão - Conjunto de dados KC2

4. Resultados

Selecionamos para este estudo aplicar quatro diferentes classificadores, selecionados por serem baseados em diferentes propriedades matemáticas.

- Naïve Bayes - *produz modelos baseados na combinação de probabilidades de uma variável associada com diferentes categorias de variáveis dependentes*
- Decision Tree - *produz modelos baseados na entropia da informação (uniformidade) de subconjuntos dos dados de treinamento obtidos pela divisão de dados baseada em variáveis independentes*
- Support Vector Machine - *produz modelos com base num híper-plano usado para separar os dados de treinamento em duas classes, onde os ítems (vetores) mais próximos do híper-plano são usados para modificar o modelo com o propósito de produzir um híper-plano com o maior distanciamento médio entre os vetores*
- Random Forest - *é uma combinação de técnicas, usando árvores de decisão, cada uma com amostras dos dados de treinamento e a decisão final tomada com base da combinação de decisões de cada árvore computando um valor modal*

Os classificadores obtiveram resultados semelhantes para todos os conjuntos de dados disponíveis, como podemos observar nas Figuras 1, 2 e 3.

Não aplicamos a análise pelo algorítimo *Support Vector Machine* ao conjunto de dados **JM1** por conta do tempo que seria necessário para executá-lo (ver Figuras 4, 5 e 6).

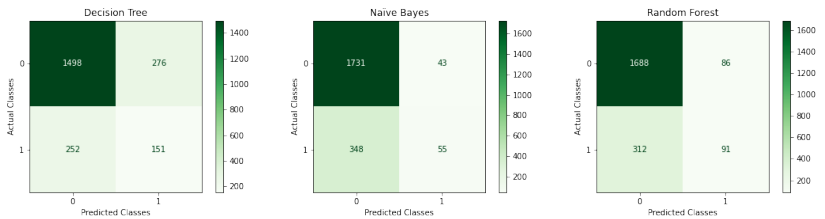

Figura 3. Matriz de Confusão - Conjunto de dados JM1

Pela análise da Matriz de Confusão, o resultado consolidado de cada conjunto de dados, *Support Vector Machine* não foi o método mais eficiente tampouco (ver Tabelas 3, 5 e 7). Nas Tabelas 2, 4 e 6 podemos observar a acurácia dos algorítimos para cada classe - *defects* - possui erros ($0 = \textit{não}, 1 = \textit{sim}$).

Algorítimo	MCC:0	MCC:1	ACC:0	ACC:1
Random Forest	0.2646763180494774	0.2646763180494774	0.94	0.94
SVM	0.2024080616510014	0.2024080616510014	0.93	0.93
Naïve Bayes	0.16585877695229945	0.16585877695229945	0.86	0.86
Decision Tree	0.18033245722931218	0.18033245722931218	0.87	0.87

Tabela 2. Acurácia - Conjunto de dados CM1

Rank	Algorítimo	Pont. Classe	Pont. Geral
1	Random Forest	6.05	2.18333
2	Support Vector Machine	5.55	1.81667
3	Naïve Bayes	4.95	1.65
3	Decision Tree	4.95	1.65

Tabela 3. Pontuação por análise da Matriz de Confusão - Conjunto de dados CM1

Algorítimo	MCC:0	MCC:1	ACC:0	ACC:1
Random Forest	0.5169843329698582	0.5169843329698582	0.8571428571428571	0.8571428571428571
SVM	0.48648427262381366	0.48648427262381366	0.8476190476190476	0.8476190476190476
Naïve Bayes	0.3587866402333297	0.3587866402333297	0.819047619047619	0.819047619047619
Decision Tree	0.3748813095095569	0.3748813095095569	0.819047619047619	0.819047619047619

Tabela 4. Acurácia - Conjunto de dados KC2

Rank	Algorítimo	Pont. Classe	Pont. Geral
1	Random Forest	7.65	3.7
2	Support Vector Machine	6.75	3.5
3	Naïve Bayes	6.35	2.38333
3	Decision Tree	6.35	2.38333

Tabela 5. Pontuação por análise da Matriz de Confusão - Conjunto de dados KC2

Algorítimo	MCC:0	MCC:1	ACC:0	ACC:1
Naïve Bayes	0.2102455048246325	0.2102455048246325	0.8203950390445567	0.8203950390445567
Decision Tree	0.2143171951455535	0.2143171951455535	0.7574644005512172	0.7574644005512172
Random Forest	0.2520032159505995	0.2520032159505995	0.8171796049609554	0.8171796049609554

Tabela 6. Acurácia - Conjunto de dados JM1

Tendo em vista a grande diferença no tempo de execução para cada tipo de algorítimo, a opção de se utilizar *Support Vector Machine* foi desconsiderada em nossa recomendação de solução para praticantes da indústria. A acurácia dos algorítimos é relativamente semelhante, o que também não é favorável para *Support Vector Machine*, e *Random Forest* mostrou-se ligeiramente superior para os conjuntos de dados menores. Já *Naïve Bayes* mostrou-se mais eficiente aplicado ao conjunto de dados **JM1** com um número maior de instâncias.

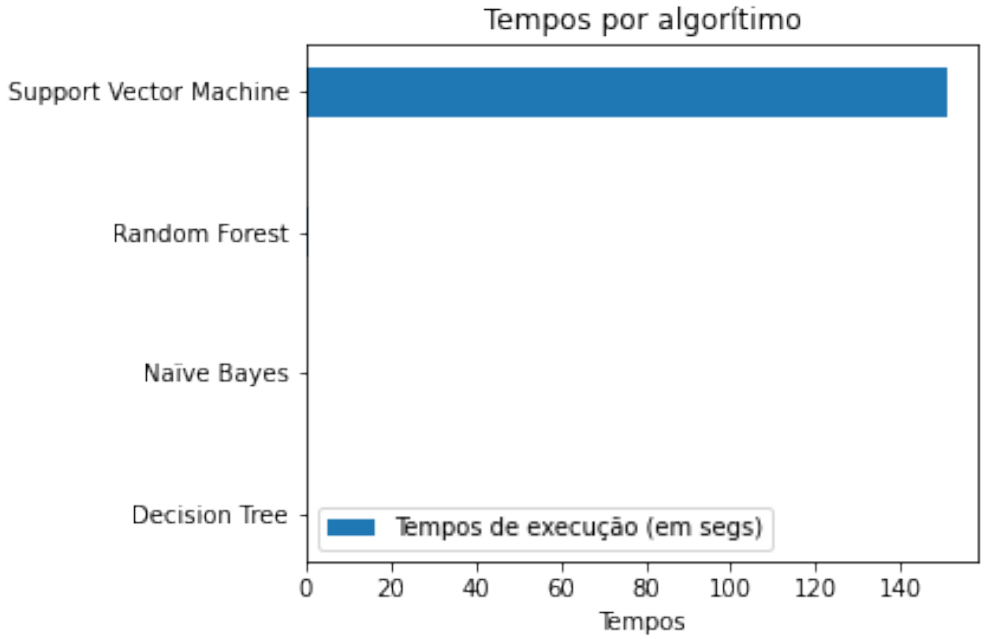

Figura 4. Tempos de execução para o conjunto de dados CM1

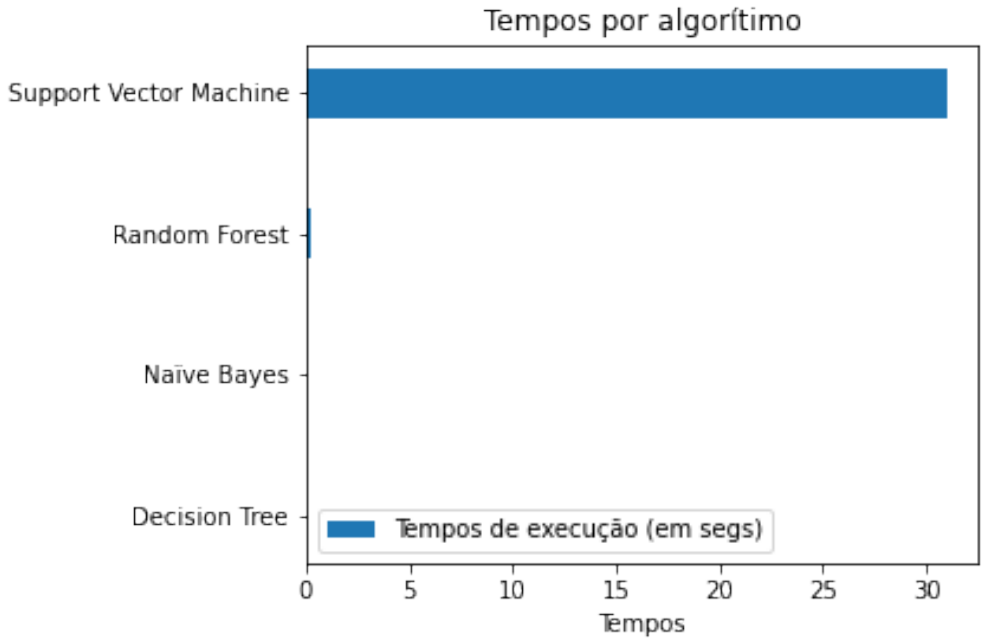

Figura 5. Tempos de execução para o conjunto de dados KC21

Rank	Algorítimo	Pont. Classe	Pont. Geral
1	Naïve Bayes	5.05	1.81667
2	Decision Tree	4.95	2.18333
3	Random Forest	4.55	2.18333

Tabela 7. Pontuação por análise da Matriz de Confusão - Conjunto de dados JM1

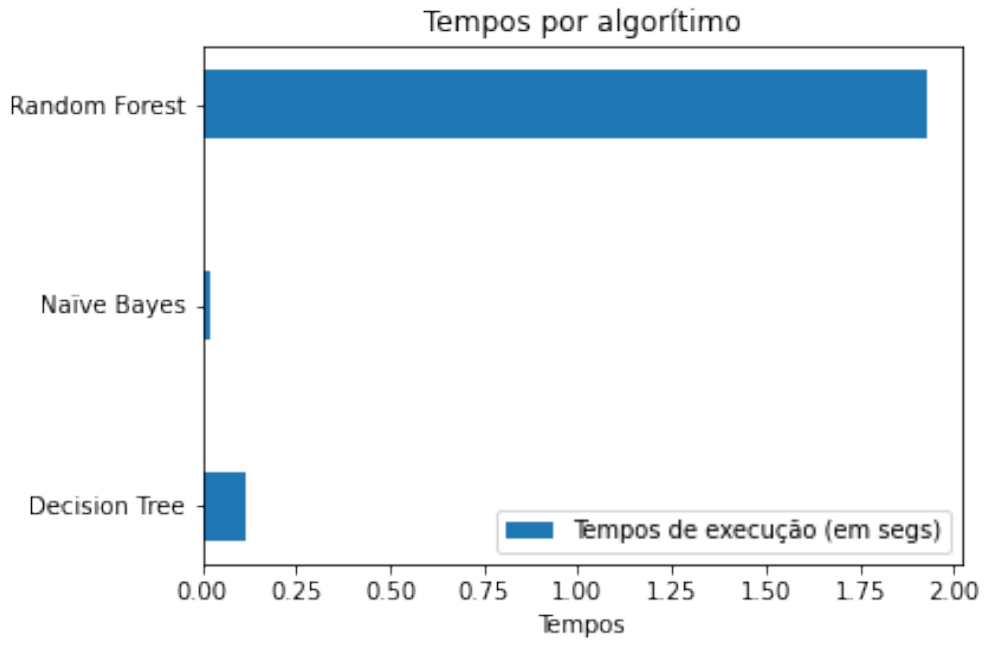

Figura 6. Tempos de execução para o conjunto de dados JM1

5. Trabalhos Relacionados

Diferentes técnicas tem sido aplicadas para prever falhas em "*software*" [Abaei et al. , Abdi et al. , Chatterjee and Maji , Dhanalaxmi et al. , Diwaker et al. , Felix , Gao et al. , Harikarthik et al. , Jayanthi and Florence , Kalsoom et al. , Kumaresan and Ganeshkumar , Mahajan et al. 2015, Panda et al. , Rani and Mahapatra , Roya et al. , Fan et al.]. Enquanto alguns autores preferem técnicas de aprendizado de máquina supervisionada [Rani and Mahapatra] ou semi-supervisionada [Abaei et al.], a maioria investiga uma abordagem híbrida utilizando conjuntamente métodos de busca evolutivos [Abdi et al. , Anwar et al. , Dhanalaxmi et al. , Diwaker et al. , Roya et al. , Shamshiri et al. , Sharifipour et al. , Tian and Gong , Varshney and Mehrotra , Yao et al.].

6. Conclusão e Trabalhos Futuros

Neste estudo, a baixa acurácia obtida pelos métodos listados nos leva a crer que abordagens que aplicam *ensembles* e as híbridas com métodos de busca sejam o próximo passo lógico a seguir. Apesar disto acreditamos que esta é uma solução simples, rápida e fácil de ser aplicada pelos praticantes, e se evitarmos utilizar *Support Vector Machine* que exigirá uma alocação de tempo e recursos maiores, pode ser uma opção para inserir este tipo de

análise num processo de desenvolvimento de *software*.

Além das abordagens utilizando *ensembles* e em conjunto com métodos de busca, o próximo passo óbvio deste trabalho é aplicá-lo na priorização dos módulos que devem ser submetidos a testes mais rigorosos, tornando assim esta uma solução interessante para incrementar a qualidade e ao mesmo tempo reduzir os custos e os prazos de confecção e execução de testes.

E finalmente, outra oportunidade que se mostra em aberto, é a possibilidade de utilizarmos as métricas de McCabe[McCabe 1976] disponíveis no conjuntos de dados utilizados e que foram negligenciadas por este estudo.

7. Siglas

MCC Coeficiente de Correlação de Matthews - do inglês: "*Matthews Correlation Coeficient*" 3

ML Aprendizado de Máquina - do inglês: "*Machine Learning*" 3

NASA Administração Nacional do Espaço e Aeronáutica - do inglês: "*National Aeronautics and Space Administration*" 4

NIST Instituto Nacional de Padrões e Tecnologia - do inglês: "*National Institute of Standards and Technology*" 2

8. Referencias

Abaei, G., Selamat, A., and Fujita, H. An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. 74:28–39.

Abdi, Y., Parsa, S., and Seyfari, Y. A hybrid one-class rule learning approach based on swarm intelligence for software fault prediction. 11(4):289–301.

Anwar, Z., Afzal, H., Bibi, N., Abbas, H., Mohsin, A., and Arif, O. A hybrid-adaptive neuro-fuzzy inference system for multi-objective regression test suites optimization. 31(11):7287–7301.

Boughorbel, S., Jarray, F., and El-Anbari, M. (2017). Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6):e0177678.

Chatterjee, S. and Maji, B. A bayesian belief network based model for predicting software faults in early phase of software development process. 48(8):2214–2228.

Dhanalaxmi, B., Naidu, G. A., and Anuradha, K. Adaptive PSO based association rule mining technique for software defect classification using ANN. 46:432–442.

Diwaker, C., Tomar, P., Poonia, R., and Singh, V. Prediction of software reliability using bio inspired soft computing techniques. 42(5):1–16.

Fan, G., Diao, X., Yu, H., Kang, Y., and Chen, L. Software defect prediction via attention-based recurrent neural network. 2019.

Felix, E. Predicting the number of defects in a new software version. 15(3):e0229131.

Fenton, N. and Pfleeger, S. (1997). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company.

Gao, R., Wong, W., Chen, Z., and Wang, Y. Effective software fault localization using predicted execution results. 25(1):131–169.

Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2011). The misuse of the nasa metrics data program data sets for automated software defect prediction. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*, pages 96–103. IET.

Haghighi, S., Jasemi, M., Hessabi, S., and Zolanvari, A. (2018). PyCM: Multiclass confusion matrix library in python. *Journal of Open Source Software*, 3(25):729.

Halstead, M. H. et al. (1977). *Elements of software science*, volume 7. Elsevier New York.

Harikarthik, S., Palanisamy, V., and Ramanathan, P. Optimal test suite selection in regression testing with testcase prioritization using modified ann and whale optimization algorithm. 22:11425–11434.

Jayanthi, R. and Florence, L. Software defect prediction techniques using metrics based on neural network classifier. 22:77–88.

Kalsoom, A., Maqsood, M., Ghazanfar, M., Aadil, F., and Rho, S. A dimensionality reduction-based efficient software fault prediction using fisher linear discriminant analysis (FLDA). 74(9):4568–4602.

Kit, E. and Finzi, S. (1995). *Software Testing in the Real World: Improving the Process*. ACM Press/Addison-Wesley Publishing Co., USA.

Kumaresan, K. and Ganeshkumar, P. Software reliability modeling using increased failure interval with ANN. 22:3095–3102.

Mahajan, R., Gupta, S. K., and Bedi, R. K. (2015). Design of software fault prediction model using BR technique. 46:849–858.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Panda, S., Munjal, D., and Mohapatra, D. A slice-based change impact analysis for regression test case prioritization of object-oriented programs. 2016.

Planning, S. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*.

Rani, P. and Mahapatra, G. S. Neural network for software reliability analysis of dynamically weighted NHPP growth models with imperfect debugging. 28(5):n/a–n/a.

Roya, P., Mahapatrab, G., and Deya, K. Neuro-genetic approach on logistic model based software reliability prediction. 42(10):4709–4718.

Ryu, D. and Baik, J. (2016). Effective multi-objective naïve bayes learning for cross-project defect prediction. *Applied Soft Computing*, 49:1062–1077.

Sayyad Shirabad, J. and Menzies, T. (2005). The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada.

Shamshiri, S., Rojas, J. M., Gazzola, L., Fraser, G., Mcminn, P., Mariani, L., and Arcuri, A. Random or evolutionary search for object-oriented test suite generation? 28(4):n/a–n/a.

Sharifipour, H., Shakeri, M., and Haghighi, H. Structural test data generation using a memetic ant colony optimization based on evolution strategies. 40:76–91.

Shepperd, M., Song, Q., Sun, Z., and Mair, C. Data quality: Some comments on the NASA software defect datasets. 39(9):1208–1215.

Tian, T. and Gong, D. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. 23(3):469–500.

Varma, T. (2000). Automated software testing: introduction, management and performance. *ACM SIGSOFT Software Engineering Notes*, 25(3):65–65.

Varshney, S. and Mehrotra, M. A hybrid particle swarm optimization and differential evolution based test data generation algorithm for data-flow coverage using neighbourhood search strategy. 42(3):417–438.

Yao, X., Gong, D., and Zhang, G. Constrained multi-objective test data generation based on set evolution. 9(4):103–108.