UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Departamento de Ciência da Computação
DCC191 -- Computação Natural
Trabalho Prático 1 - Programação Genética
Guilherme de Abreu Lima Buitrago Miranda - 2018054788

1 - Introdução

O trabalho prático 1 da disciplina de Computação Natural tem como objetivo o desenvolvimento de uma solução para problemas usando Programação Genética (GP). Em particular, o algoritmo a ser desenvolvido busca resolver o problema de regressão simbólica, em que, dado um conjunto m de amostras provenientes de uma função desconhecida f: R^n -> R, representados por uma dupla <X, Y>, onde X pertence a $R^{m \times n}$ e Y pertence a R^m, quer-se encontrar a função f que melhor atende às amostras analisadas.

Abaixo, encontram-se detalhes relativos à implementação do trabalho, experimentos, conclusões e referências bibliográficas.

2 - Implementação

Devido à natureza dos problemas a serem resolvidos no trabalho (como clustering, por exemplo), decidiu-se pela adoção da linguagem Python. Além disso, famosas bibliotecas do mundo Python foram empregadas, tais como Numpy, Pandas Sklearn e PyClustering.

A primeira ação a ser executada pelo algoritmo é a leitura do dataframe por meio do Pandas. Em seguida, são definidos os terminais e as operações (funções) matemáticas que podem ter um indivíduo. Para representar os indivíduos, várias abordagens foram consideradas. Contudo, após a leitura do material didático fornecido, optou-se pela adoção de uma árvore para tal representação, ainda que essa escolha potencialmente tenha impactos negativos na performance do algoritmo. Assim, os nós terminais (folhas) da árvore são os parâmetros do dataset, enquanto os nós intermediários são compostos pelas operações previamente mencionadas.

Em seguida, a população é inicializada utilizando o método Ramped Half-and-Half, tendo uma altura variando de 2 até 7. Com 6 alturas distintas e 2 tipos de inicialização (devido ao método empregado), fica evidente que o total da população inicial é sempre múltiplo de 12 (o que explicará, posteriormente, alguns dados escolhidos para a análise experimental).

Para o cálculo da fitness dos indivíduos gerados, em primeiro lugar, é necessário que o algoritmo seja capaz de extrair sua expressão do indivíduo. Assim, é criada a função evaluate, que caminha na árvore de maneira pré-ordem e gera uma lista com operadores e operandos. Os cálculos são feitos à medida que tornam-se possíveis, e no término de sua execução, apenas o resultado final é retornado. Tal função é passada como parâmetro para o algoritmo kmeans do pyclustering e, ao final, o conjunto desses indivíduos é submetido ao v_measure_score do sklearn.

Posteriormente, é necessário que tais indivíduos passem por operações genéticas, como o crossover e a mutação, implementados no arquivo genetic_methods.py a fim de conferir uma maior organização ao código. Para ambos os métodos, existe também um método acompanhante, que busca caminhar na árvore que representa o indivíduo e encontrar o nó a ser cruzado ou mutado. Em comum, ambos os métodos invocam a função get_node_and_path que, conforme o próprio nome explicita, obtém um nó da árvore e seu respectivo caminho.

Assim, para a mutação, um nó é escolhido ao acaso e seu conteúdo é trocado por um equivalente (terminais só são trocados por outros terminais e funções por outras funções). Já no caso do cruzamento, é escolhido um nó aleatório do primeiro pai e, para o segundo pai, é necessário que a profundidade desse nó, somada a do nó anterior, não ultrapasse a máxima da árvore. Dessa forma, é impossível haver um crescimento desordenado dos indivíduos.

Por fim, ainda sobre os métodos genéticos, há o tournament_selection. Nele, após utilizar do NumPy para aleatorizar os indivíduos, são retornados: ou o indivíduo com maior fitness entre os k primeiros, ou, no caso do cruzamento, o indivíduo com o maior fitness entre os k primeiros e outro com a maior fitness entre os k últimos da lista.

Dessa forma, retomando o fim do quarto parágrafo desta seção, os indivíduos recém construídos e com sua fitness calculada entram num loop for, definido pelo número de gerações requeridas. Assim, enquanto o tamanho da nova geração não

for igual ao da anterior, são executados os algoritmos genéticos previamente mencionados de acordo com probabilidades definidas (e devidamente variadas, conforme será explicado na próxima seção). Ao término da execução da repetição mais interna, a fitness dos novos indivíduos é calculada, assim como o melhor indivíduo da população é copiado para a próxima.

Por fim, o melhor indivíduo de todas as gerações é testado contra os dados de teste dos datasets fornecidos e seu resultado é reportado.

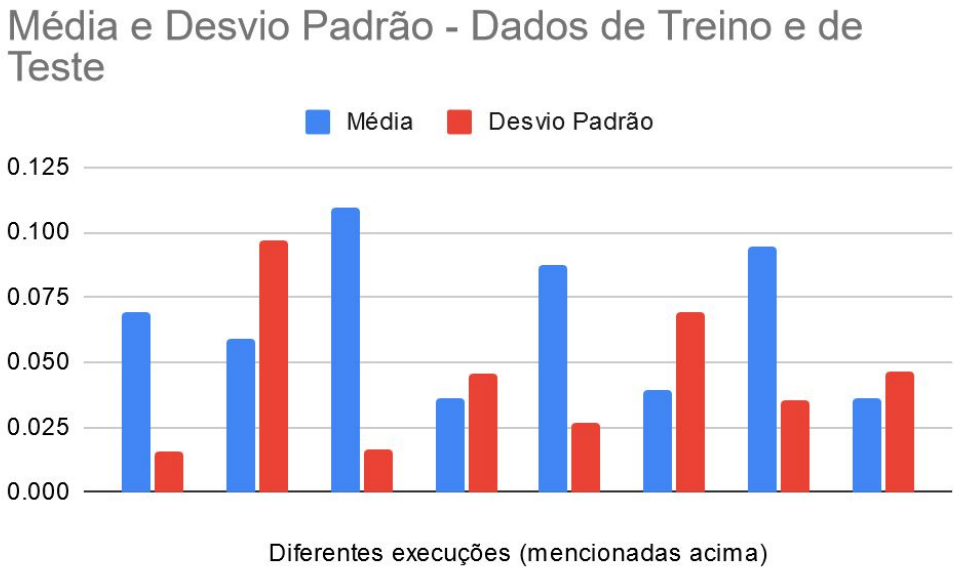
3 - Experimentos

Para a realização dos experimentos, foram utilizados os seguintes parâmetros:

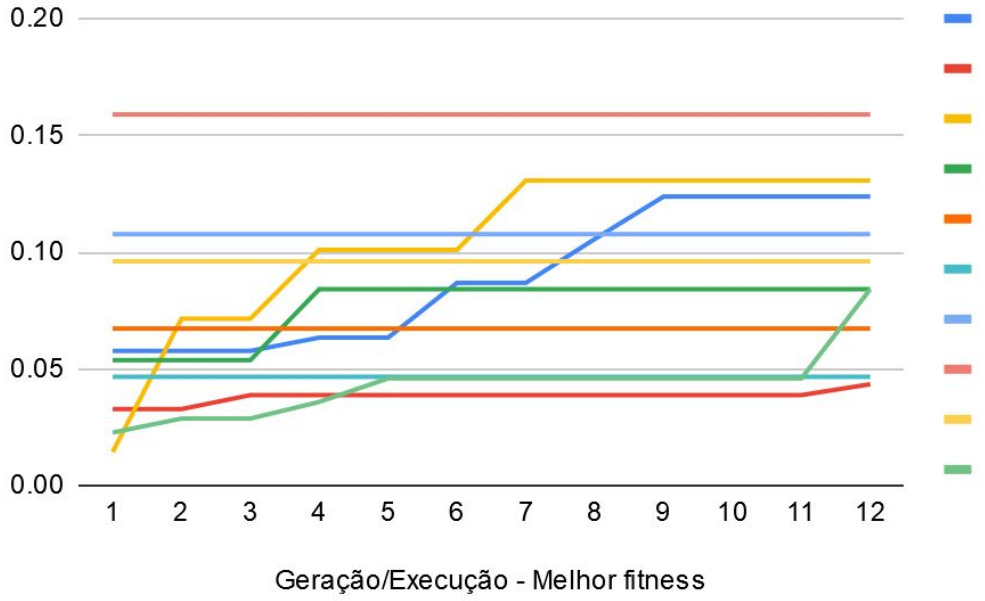
- Dataset: breast_cancer_coimbra
- Indivíduos e Gerações: 12, 24, 36 e 48
- Probabilidade de crossover e mutação: 0.09 e 0.05, 0.6 e 0.3
- Valor k do torneio: 2 e 5, 3 e 7 (valores menores para 12 e 24 indivíduos/gerações, valores maiores para 36 e 48 indivíduos/gerações)

Consoante ao enunciado do trabalho, cada uma das combinações acima mencionadas foi executada 10 vezes, a fim de obter-se a média e o desvio padrão da fitness alcançada nos dados de treino e de teste. Assim, temos:
Para 12 indivíduos, da esquerda para a direita no gráfico:

- 12 Indivíduos / 12 Gerações / 0.9 mut / 0.05 cross / 2 torneio
- 12 Indivíduos / 12 Gerações / 0.9 mut / 0.05 cross / 5 torneio
- 12 Indivíduos / 12 Gerações / 0.6 mut / 0.3 cross / 2 torneio
- 12 Indivíduos / 12 Gerações / 0.6 mut / 0.3 cross / 5 torneio

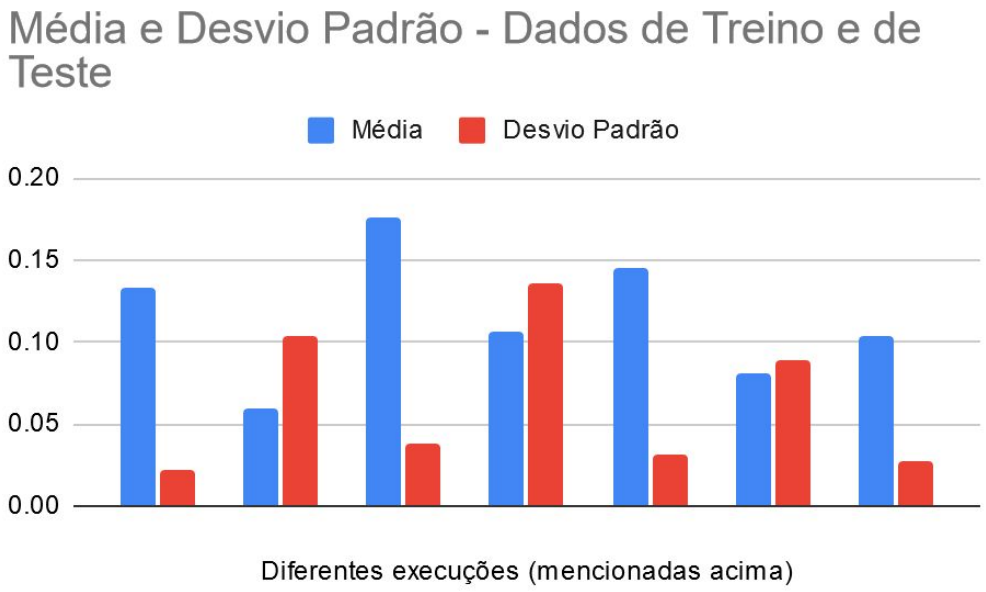


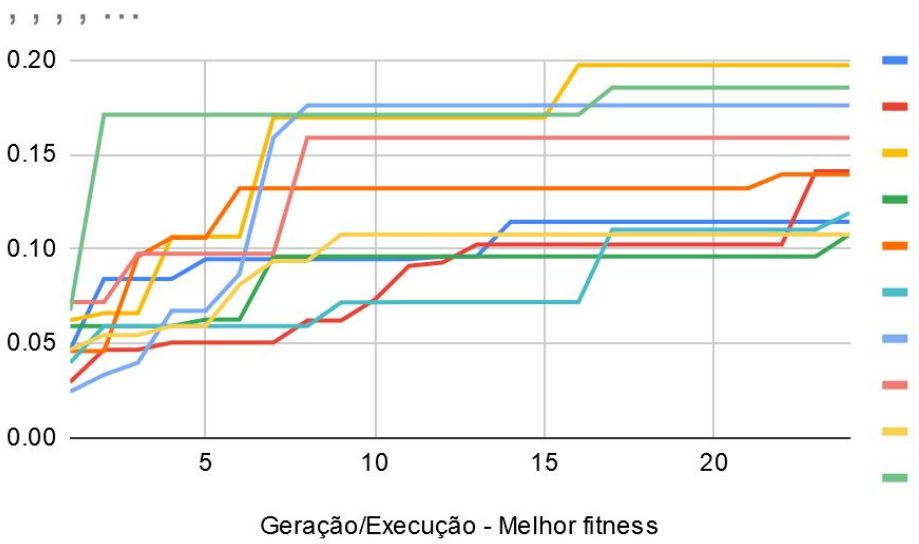
A melhor combinação se comportou da seguinte forma (durante as 10 execuções):



Para 24 indivíduos, da esquerda para a direita no gráfico (gráfico da melhor combinação logo em sequência):

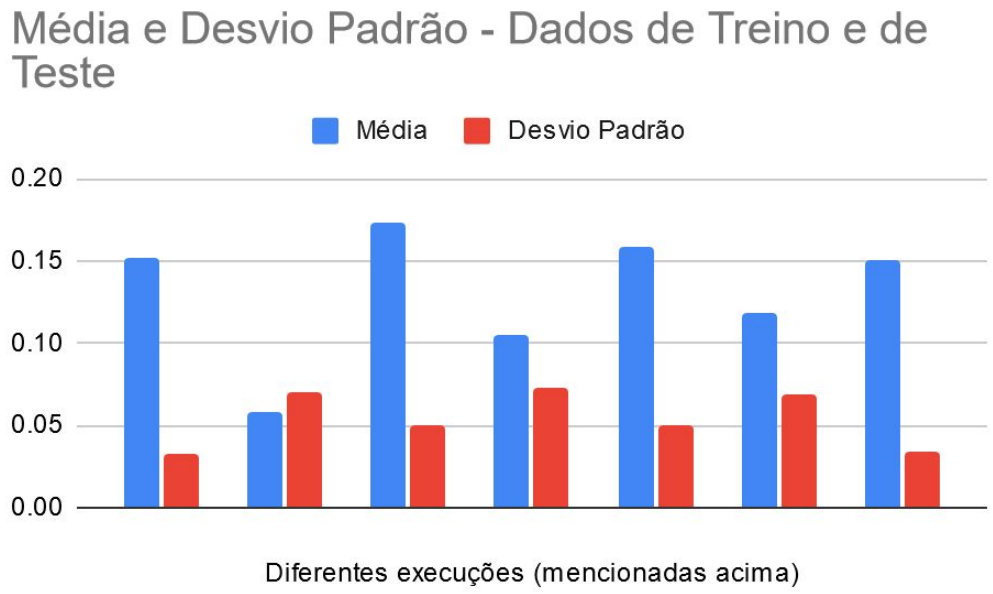
- 24 Indivíduos / 24 Gerações / 0.9 mut / 0.05 cross / 2 torneio
- 24 Indivíduos / 24 Gerações / 0.9 mut / 0.05 cross / 5 torneio
- 24 Indivíduos / 24 Gerações / 0.6 mut / 0.3 cross / 2 torneio
- 24 Indivíduos / 24 Gerações / 0.6 mut / 0.3 cross / 5 torneio

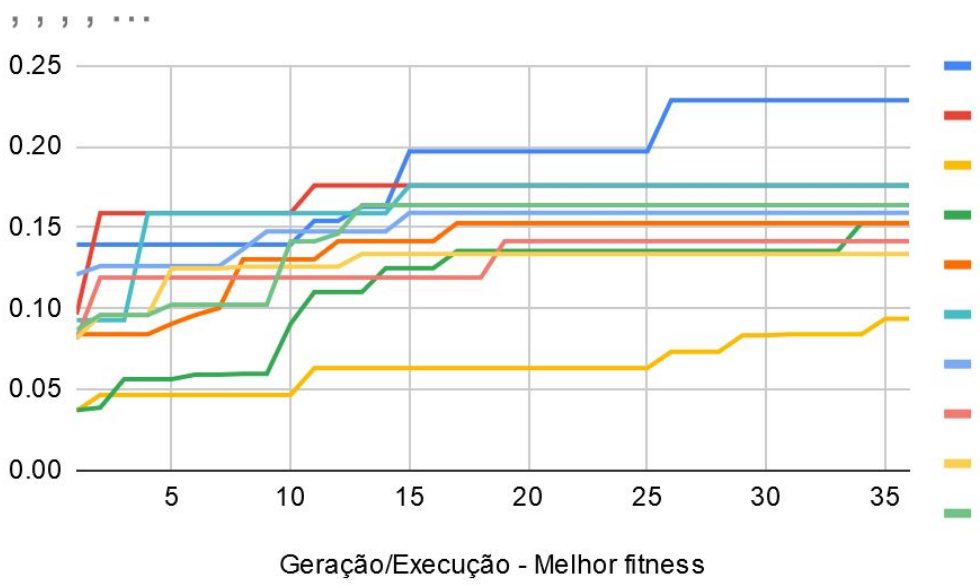




Para 36 indivíduos, da esquerda para a direita no gráfico (gráfico da melhor combinação logo em sequência):

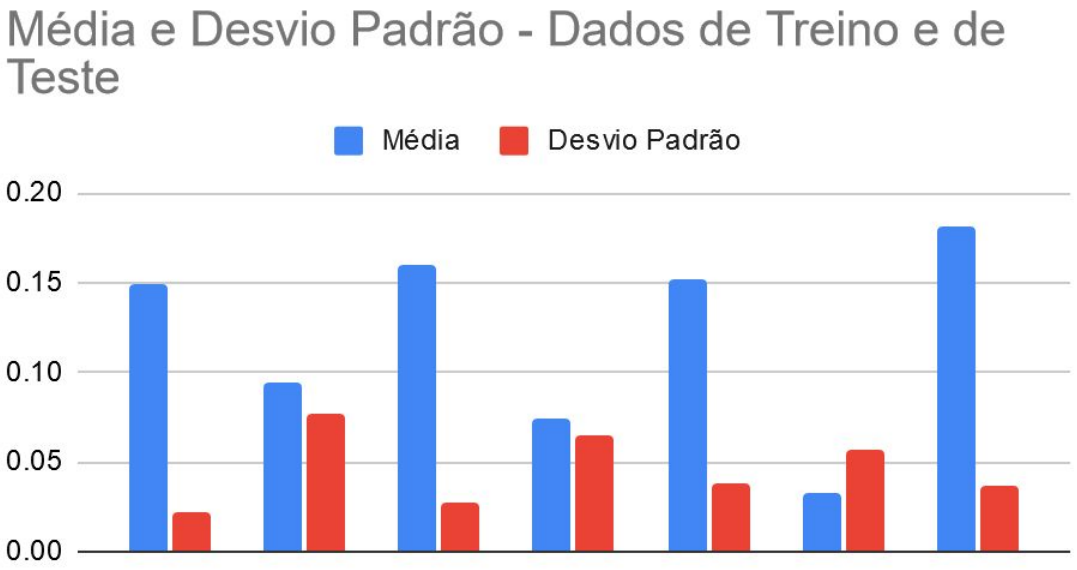
- 36 Indivíduos / 36 Gerações / 0.9 mut / 0.05 cross / 3 torneio
- 36 Indivíduos / 36 Gerações / 0.9 mut / 0.05 cross / 7 torneio
- 36 Indivíduos / 36 Gerações / 0.6 mut / 0.3 cross / 3 torneio
- 36 Indivíduos / 36 Gerações / 0.6 mut / 0.3 cross / 7 torneio

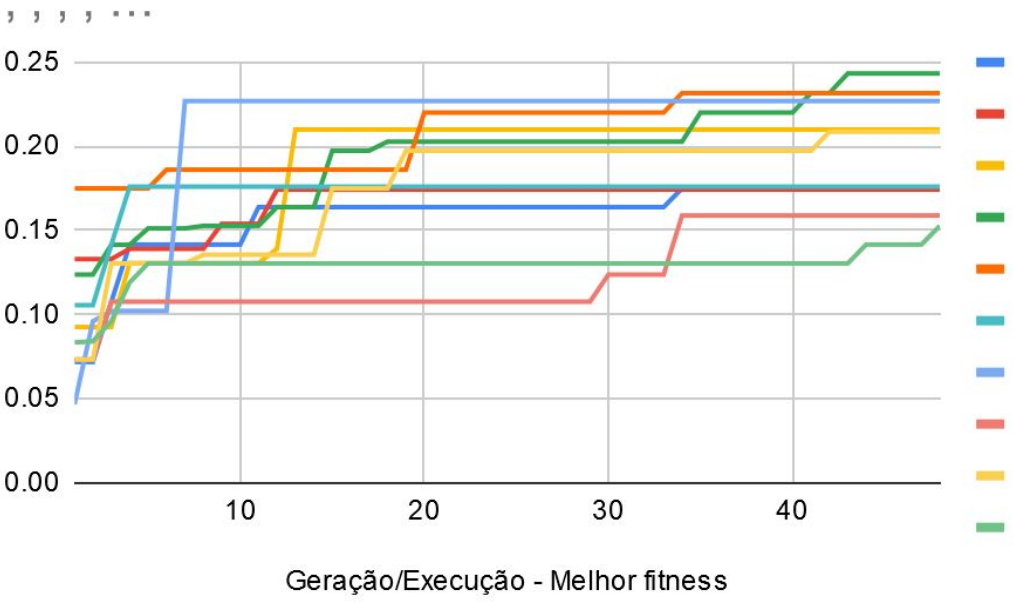




Para 48 indivíduos, da esquerda para a direita no gráfico (gráfico da melhor combinação logo em sequência):

- 48 Indivíduos / 48 Gerações / 0.9 mut / 0.05 cross / 3 torneio
- 48 Indivíduos / 48 Gerações / 0.6 mut / 0.3 cross / 7 torneio
- 48 Indivíduos / 48 Gerações / 0.6 mut / 0.3 cross / 3 torneio
- 48 Indivíduos / 48 Gerações / 0.9 mut / 0.05 cross / 7 torneio





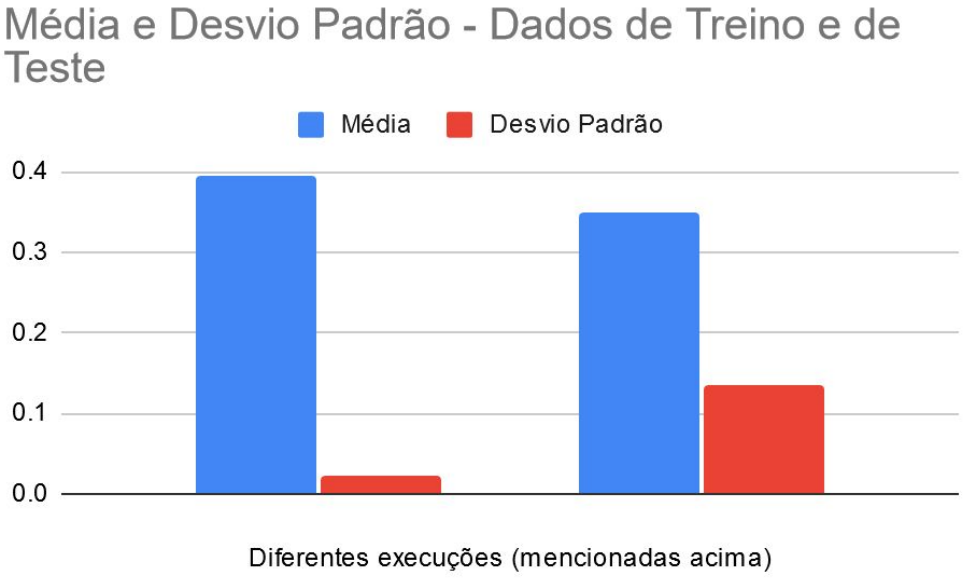
A partir dos dados e dos gráficos apresentados, algumas conclusões interessantes podem ser observadas. Nas execuções menores, conforme se aumenta o número de gerações/indivíduos, de maneira geral, melhor é o resultado obtido, tanto nos dados de treino, quanto nos dados de teste. Contudo, a partir de aproximadamente 32 a 36 gerações, a melhora apresentada é bastante diminuta.

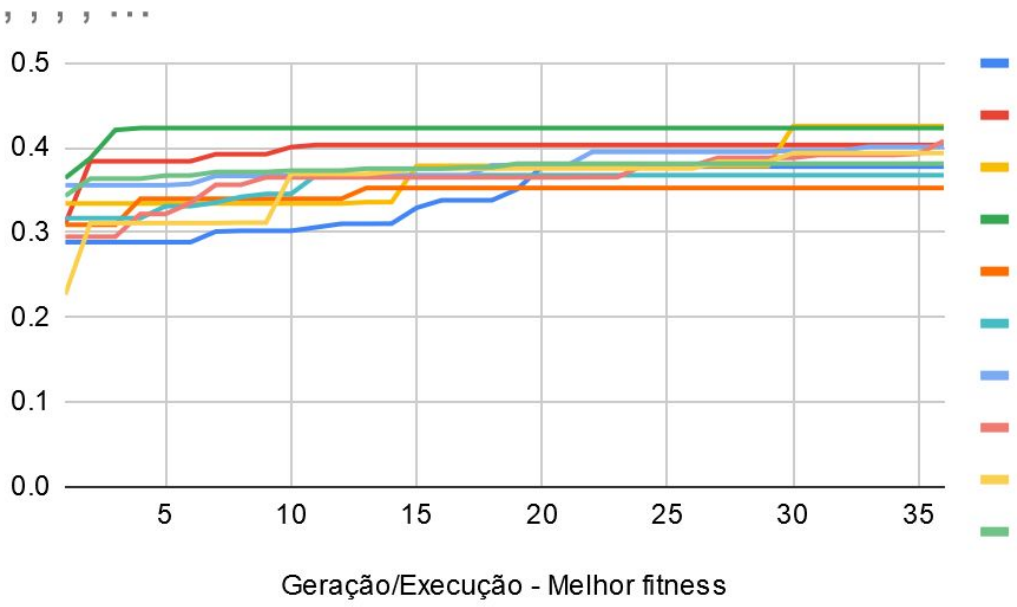
Assim, considerando o tradeoff entre tempo de processamento e melhora no resultado final, crê-se que o melhor custo-benefício foi encontrado na terceira

execução. Em particular, o intuito inicial era executar o algoritmo para um número maior de gerações e indivíduos, como 60 ou 120, mas a demora no processamento de instâncias tão grandes inviabilizou a análise.

A variação de outros parâmetros, como a probabilidade do crossover e da mutação, assim como a diferenciação no torneio mostrou uma melhora marginal, ou, em alguns casos, variações inconclusivas.

Por fim, executou-se, no dataset glass, o algoritmo com os parâmetros considerados mais interessantes (36 Indivíduos / 36 Gerações / 0.6 mut / 0.3 cross / 7 torneio). Obtém-se, portanto, os seguintes resultados:





Observa-se, dessa forma, que a combinação de parâmetros também alcança resultados interessantes em outras bases de dados.

4 - Conclusões

Para além da prática com a linguagem Python e suas bibliotecas, a implementação do trabalho foi bastante proveitosa para compreender efetivamente os conceitos vistos até o presente momento na disciplina. O aprendizado se deu, sobretudo, no momento de transpor os conceitos vistos em aula para o código propriamente dito.

Além disso, a análise experimental também contribuiu muito para refletir a respeito de detalhes da programação genética, assim como outras dúvidas dos colegas apresentadas no fórum de discussões. Com isso, considera-se que o trabalho foi de grande proveito para o processo de aprendizagem.

5 - Bibliografia

- Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd.
- pyclustering Distance Metric Class Reference - https://pyclustering.github.io/docs/0.9.0/html/df/df9/classpyclustering_1_1utils_1_1metric_1_1distance__metric.html
- Count duplicate lists inside a list - https://stackoverflow.com/questions/44747524/count-duplicate-lists-inside-a-list