Documentação Trabalho Prático I de Computação Natural: Regressão Simbólica usando Programação Genética

Ronald Davi Rodrigues Pereira
ronald.pereira@dcc.ufmg.br

Universidade Federal de Minas Gerais

29 de Setembro de 2018

1 Introdução

Em programação genética, uma aplicação muito recorrente é a de se encontrar uma função que melhor represente um número determinado de pontos em um conjunto de dados, de modo a se reduzir ao máximo o erro[1]. Geralmente, não são dadas entradas específicas para determinado algoritmo, fazendo-o ter que explorar um determinado número de possíveis soluções e avaliar quais são as melhores para se evoluir e tentar chegar à uma aproximação da solução.

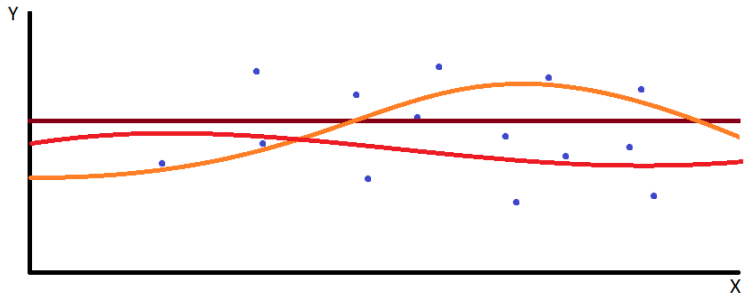

Figura 1 – Exemplo ilustrativo de uma regressão simbólica

Como qualquer outro algoritmo probabilístico[2], não é garantido que essa solução ótima será alcançada. Porém, por meio de heurísticas e métodos empíricos, é possível se garantir que pelo menos uma aproximação satisfatória da solução ótima foi alcançada, caso sejam utilizados parâmetros que, em troca de tempo de execução e complexidade, reduzem ao máximo o erro das suas possíveis soluções.

O objetivo desse trabalho é de resolver o problema de regressão simbólica utilizando programação genética. Portanto, buscamos uma solução que se aproxime da solução ótima do problema proposto através da evolução de uma população inicial randômica. Vários métodos empíricos podem ser utilizados para a otimização desses algoritmos e isso será discutido e mostrado nesse documento para algumas bases de dados específicas.

2 Implementação

2.1 Descrição dos Arquivos

symbolic_regression.py Programa principal que realiza o controle do fluxo de execução do programa

best_individual.py Biblioteca customizada que guarda o melhor indivíduo geral de uma execução do programa ao longo das gerações

data.py Biblioteca customizada que realiza a leitura, processamento, rotulamento e armazenamento das entradas (conjuntos de treino e teste)

fitness.py Biblioteca customizada que realiza os cálculos de *fitness* relacionados a cada indivíduo da população

individual.py Biblioteca customizada que realiza a criação de novos indivíduos na população com o auxílio de outras bibliotecas customizadas e o armazena

operators.py Biblioteca customizada que implementa os operadores genéticos (mutação, cruzamento e reprodução)

protected_math_functions.py Biblioteca customizada que implementa as funções que devem ser protegidas para a corretude dos cálculos e execução do algoritmo

selection.py Biblioteca customizada que implementa as funções de seleção (no caso, torneio) e de pegar o melhor indivíduo da geração atual

statistics.py Biblioteca customizada que implementa o cálculo e a impressão de estatísticas ao longo das gerações numa execução do algoritmo

tree.py Biblioteca customizada que implementa a estrutura de dados do tipo árvore e a classe de geração randômica de árvores

Makefile Arquivo para auxílio da execução do programa

requirements.txt Arquivo que explicita todas as bibliotecas externas (não-padrões do Python 3) utilizadas na confecção do trabalho, bem como as suas versões

tests.sh Arquivo interativo para execução de testes repetidos automaticamente, bem como a separação em diferentes pastas das saídas do programa

input/ Pasta que contém todas as entradas (conjuntos de treino e teste)

output/ Pasta que contém todos os testes executados

2.2 Estatísticas

Ao final de cada geração, são impressas as seguintes estatísticas:

Generation Geração atual

Best Individual Fenótipo do melhor indivíduo na geração atual

Best Individual Fitness *Fitness* do melhor indivíduo na geração atual

Worst Individual Fitness *Fitness* do pior indivíduo na geração atual

Mean Fitness Fitness média da população na geração atual

Crossover child worse than fathers' mean Quantidade de filhos em que a sua *fitness* foi pior que a *fitness* média dos pais

Crossover child better than fathers' mean Quantidade de filhos em que a sua *fitness* foi melhor que a *fitness* média dos pais

Repeated individuals Quantidade de indivíduos que são cópia de outros indivíduos na população

Unique individuals Quantidade de indivíduos que são únicosna população (mesmo que possuam cópias, o primeiro indivíduo é único)

Como por exemplo:

```
***Generation: 3 ***
Best Individual: (prot_math.log(prot_math.sqrt(((math.cos(prot_math.sqrt(X[0])))
/prot_math.div(((X[0])**prot_math.pow(X[0]))-((X[0])-(-18.049353628513387))))-
(prot_math.log2(((-27.64699771826811)-(16.860974200669702))*
(math.sin(-81.63376465651635)))))))
Best Individual Fitness: 1.0631915540949075
Worst Individual Fitness: 1.2134019899750847
Mean Fitness: 1.0872252238357376
Crossover child worse than fathers' mean: 4
Crossover child better than fathers' mean: 12
Repeated individuals: 20
Unique individuals: 80
```

Ao final da execução do algoritmo (quando o número de gerações é alcançado), são impressas as seguintes estatísticas:

Best Individual Fenótipo do melhor indivíduo encontrado ao longo de todas as gerações

Best Individual Train Fitness *Fitness* do melhor indivíduo encontrado ao longo de todas as gerações no conjunto de treino

Best Individual Test Fitness *Fitness* do melhor indivíduo encontrado ao longo de todas as gerações no conjunto de teste

Como por exemplo:

```
***TEST STATISTICS***
Best Individual: ((((X[1])-((X[0])+(math.cos(prot_math.log10(X[1]))))))-
((X[0])+(math.cos(prot_math.log10(X[1]))))))-((-13.710583669730525)+
(math.sin((X[1])*((math.sin(X[1]))**prot_math.pow((X[0])**
prot_math.pow(-25.23268751116852)))))))
Best Individual Train Fitness: 0.9874143556195495
Best Individual Test Fitness: 0.9763240330360526
```

2.3 Fitness

Para o cálculo da *fitness*, foi utilizado uma função de raiz quadrada do erro quadrático médio normalizada (NRMSE), que pode ser expressa como:

$$fitness(Ind) = \sqrt{\frac{\sum_{i=1}^{N}(y_i - eval(Ind, x_i))^2}{\sum_{i=1}^{N}(y_i - \frac{1}{N} * \sum_{i=1}^{N}(y_i))^2}}$$

onde N é o número de instâncias fornecidas, Ind é o indivíduo sendo avaliado, $eval(Ind, x_i)$ avalia o indivíduo Ind na i-ésima instância de X e y_i é a saída esperada para a entrada x_i.

2.4 Decisões de Implementação

Serão demonstradas abaixo todas as decisões de implementação que foram possíveis de serem tomadas durante o desenvolvimento desse algoritmo, de forma a explicitar quais estratégias foram utilizadas por mim nesse trabalho.

2.4.1 Representação do Indivíduo

Um indivíduo nesse problema representa sempre uma solução válida, ou seja, uma configuração de operações matemáticas que constituem uma função. Para tal representação desses indivíduos, foi escolhido uma abordagem clássica em Programação Genética: a representação por árvores.

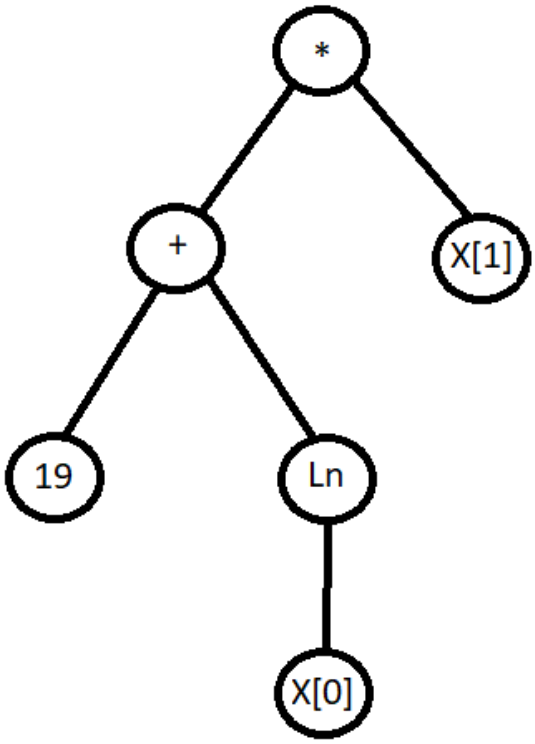

Figura 2 – Exemplo ilustrativo da representação de um indivíduo (Genótipo)

Portanto, conforme podemos observar na figura acima, um indivíduo é representado por uma árvore aonde seus nós internos são constituídos por funções matemáticas, podendo ser tanto funções binárias - requerem dois operandos - quanto funções unárias - requerem somente um operando -, e suas folhas são constituídas por terminais, podendo ser tanto constantes quanto variáveis. Esses indivíduos são gerados aleatoriamente de acordo com certas estratégias que serão discutidas no próximo tópico.

Para as funções binárias, foram utilizadas operações matemáticas de soma, subtração, multiplicação, divisão protegida - caso o denominador seja zero, retorna um número muito próximo de 0 (0.00000000000000001, especificamente nessa implementação) como novo denominador, de forma evitar erros de divisão por 0 - e potenciação protegida - caso a potência não esteja entre $(-1, 1)$, retorna 0, e caso esteja, retorna a aproximação inteira dele, de modo a evitar potenciações que gerem erro por serem equivalentes a uma radiciação de um número -.

Já para as funções unárias, foram utilizadas operações matemáticas de seno, cosseno, raiz quadrada protegida - caso o radicando seja negativo, é retornado o valor 0, de modo a evitar número imaginários na representação do indivíduo -, logaritmo natural protegido, logaritmo na base 2 protegido e logaritmo na base 10 protegido - caso o logaritmando seja ≤ 0, é retornado o $\log_y(0.00000000000000001)$, sendo y a base do referido logaritmo, de modo a evitar erros matemáticos -.

Por último, para os terminais, foram utilizadas constantes do tipo *float* que pertencem à uma distribuição uniforme no intervalo $[-100, 100)$ e variáveis do tipo X_i, sendo i o índice referente à coluna dada na entrada.

2.4.2 Geração da População Inicial

A geração da população inicial é partida de escolhas randômicas de utilização de funções ou terminais. Para aumentar o nível de diversidade na população inicial, foi utilizado a estratégia de montar populações *ramped-half-and-half* [3]. Desse modo, metade da população é gerada a partir do método *grow* - que consiste em gerar uma árvore não-balanceada de altura máxima h, em que os terminais podem estar em diferentes alturas - e a outra metade a partir do método *full* - que consiste em gerar uma árvore balanceada de altura exata h, em que todos os terminais têm de estar nessa altura h -.

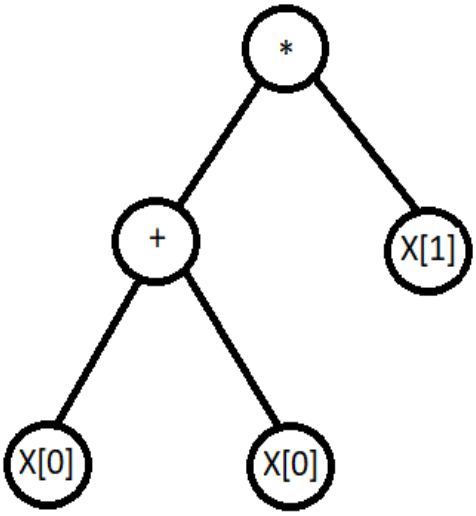

Figura 3 – Exemplo ilustrativo da representação de um indivíduo utilizando o método Grow

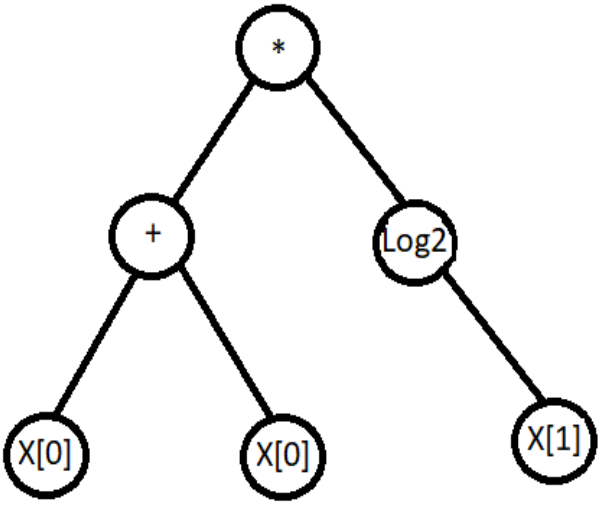

Figura 4 – Exemplo ilustrativo da representação de um indivíduo utilizando o método Full

2.4.3 Operadores Genéticos

Para esse trabalho, foram utilizados três diferentes operadores genéticos: mutação, cruzamento e reprodução com o objetivo de gerar modificações aleatórias na população, gerando outros indivíduos (mutação e cruzamento), ou até mesmo de replicar um indivíduo na próxima geração (reprodução). Esses operadores serão explicados separadamente abaixo.

2.4.3.1 Mutação

A mutação acontece em um indivíduo de forma a variar aleatoriamente uma sub-árvore, gerando um outro indivíduo possivelmente distinto de seu pai. Caso o parâmetro de operadores elitistas esteja ativado, esse filho gerado pela mutação só será passado para

a próxima geração caso seja melhor que o pai em relação à sua *fitness* ($fitness_{filho} < fitness_{pai}$), nesse caso, pois se trata de um problema de minimização). Caso contrário, esse filho gerado será levado automaticamente para a próxima geração, não importando a sua *fitness*.

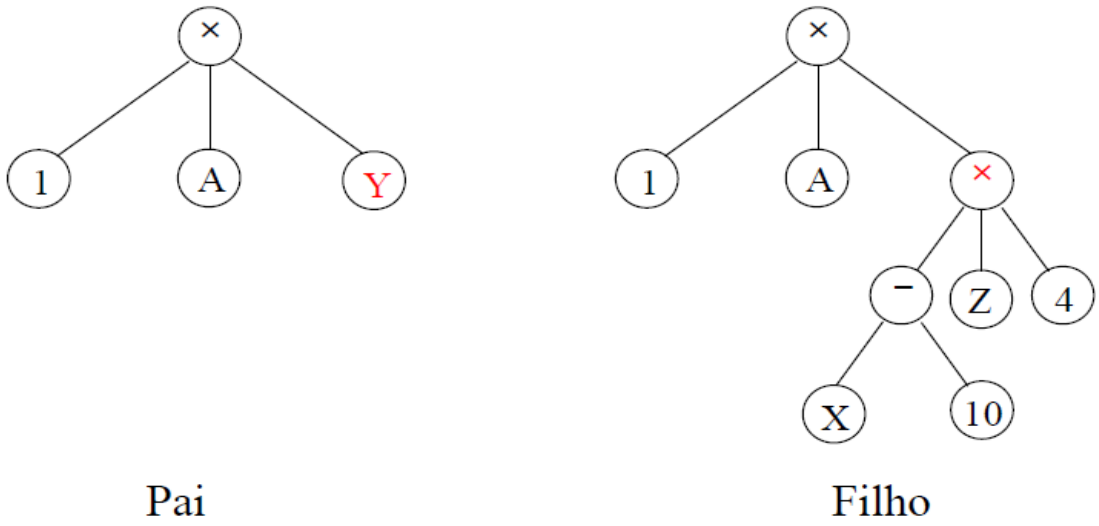

Figura 5 – Exemplo ilustrativo de uma operação de mutação em uma árvore

Esse operador foi implementado para gerar uma maior aleatoriedade na população, de modo a expandir o espaço de busca pelas soluções possíveis do problema - esse termo geralmente é conhecido como *exploration* -. Desse modo, soluções viáveis que estejam relativamente distantes uma da outra podem ser consideradas e uma possível melhor solução para uma execução do algoritmo.

2.4.3.2 Cruzamento

O cruzamento acontece entre dois indivíduos de forma a trocar entre eles uma sub-árvore escolhida de forma aleatória, gerando dois indivíduos possivelmente distintos de seus pais. Caso o parâmetro de operadores elitistas esteja ativado, esses filhos gerados pelo cruzamento só serão passados para a próxima geração caso algum deles seja melhor que algum pai em relação a sua *fitness*. Desse modo, os indivíduos são ranqueados em ordem crescente de suas *fitness* e apenas os dois melhores (que possuem as duas menores *fitness*, pois se trata de um problema de minimização) passam para a próxima geração. Caso contrário, os dois filhos gerados pelo cruzamento sempre passam para a próxima geração, não importando os valores de suas *fitness*.

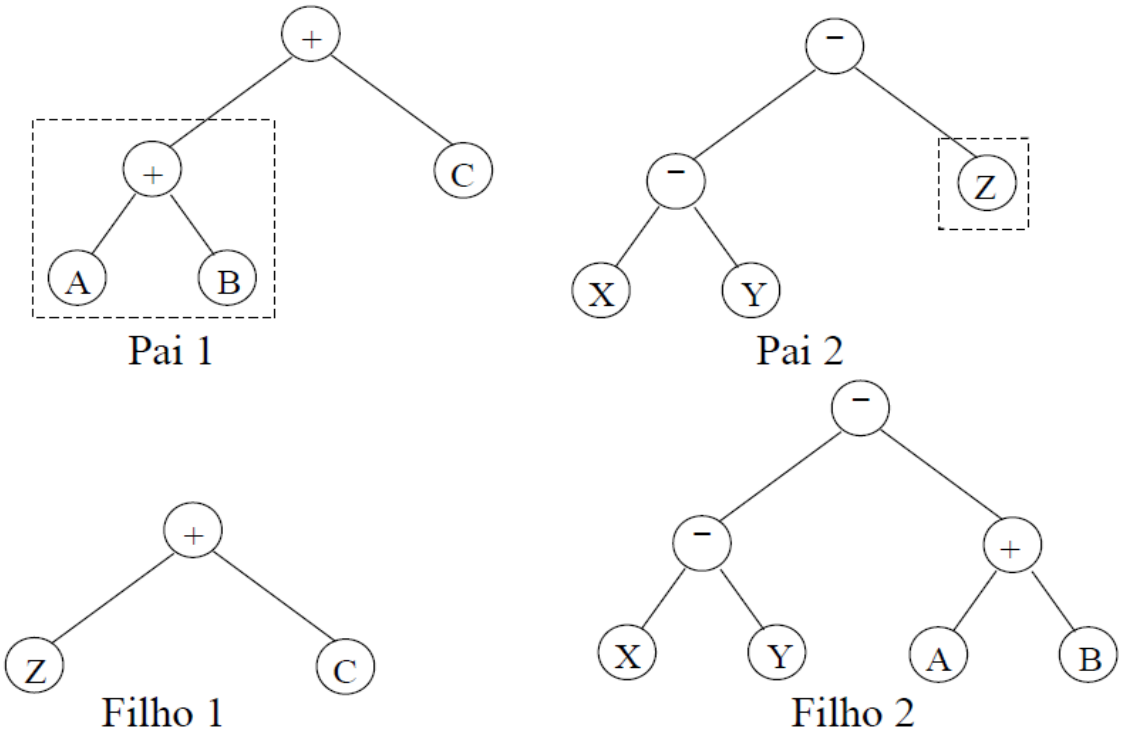

Figura 6 – Exemplo ilustrativo de uma operação de cruzamento entre duas árvores

Esse operador foi implementado para gerar uma combinação aleatória de possíveis soluções, de modo a restringir o espaço de busca pelas soluções possíveis do problema - esse termo geralmente é conhecido como *exploitation* -. Desse modo, é possível se convergir para uma solução viável gerada do cruzamento entre indivíduos que foram considerados como uma solução candidata para o problema em uma determinada execução do algoritmo.

2.4.3.3 Reprodução

Esse operador foi implementado para somente ser o complemento das probabilidades de mutação e cruzamento ($P_{reprodução} = 1 - P_{mutação} - P_{cruzamento}$). Ele somente replica na próxima geração um indivíduo aleatório. Uma possível contribuição desse operador é de aumentar a diversidade na próxima população caso um pai seja substituído pelo seu filho em algum dos operadores genéticos citados anteriormente, colocando-o na próxima geração.

2.4.4 Parâmetros e Execução

As variações de parâmetros foram implementadas de forma a facilitar testes e execuções distintas do algoritmo. Todos os parâmetros tidos como minimamente importantes foram externalizados em um comando do tipo *make* para execução do programa principal com esses parâmetros passados por linha de comando.

Abaixo, um exemplo de uma execução com todos os parâmetros:

```
# Um menu de ajuda pode ser visualizado executando o comando make help
make run POP=50 GEN=20 MUT=0.05 CROSS=0.9 K=2 SEED=1 ELIT=1 ELITOPS=1
TRAIN=./input/datasets/synth1/synth1-train.csv
TEST=./input/datasets/synth1/synth1-test.csv
```

POP Tamanho da população (*integer* $\in [1, \infty)$)

GEN Quantidade de gerações totais a serem executadas (*integer* $\in [1, \infty)$)

MUT Probabilidade de mutação (*float* $\in [0, 1]$)

CROSS Probabilidade de cruzamento (*float* $\in [0, 1]$)

Tenha sempre em mente que $P_{mutação} + P_{cruzamento} \leq 1$

K Tamanho do torneio (*integer* $\in [1, \infty)$)

SEED Ativar semente para funções *random* (*boolean* $\in 0, 1$)

ELIT Ativar elitismo (*boolean* $\in 0, 1$)

ELITOPS Ativar operadores elitistas (*boolean* $\in 0, 1$)

TRAIN Caminho relativo ao arquivo referente ao conjunto de treinamento do tipo CSV

TEST Caminho relativo ao arquivo referente ao conjunto de teste do tipo CSV

2.4.5 Comparação de Indivíduos Repetidos

Para a comparação de indivíduos repetidos foi preferido uma implementação mais simples. Ela acontece a partir de dois indivíduos que são iguais somente se suas árvores forem exatamente as mesmas. Essa abordagem foi escolhida pela simplicidade dessa comparação, pois uma outra possível comparação seria por comparar todos os resultados obtidos pelo conjunto de teste e, caso eles mantenham o valor e a ordem entre dois indivíduos, eles são iguais. Porém, essa segunda implementação se demonstrou muito lenta para conjunto de dados relativamente grandes, tendo uma complexidade de tempo de $O(N^2 * T)$, sendo N o número de indivíduos na população e T o número de linhas no conjunto de treino.

Já para a implementação realizada, essa complexidade de tempo é de $O(N^2 * h)$, sendo N o número de indivíduos na população e h a altura máxima da árvore, pois como o caminhando é feito *in-order*, a complexidade de tempo de se caminhar nessa árvore é de $O(h)$. Portanto, nesse caso, como a altura máxima da árvore h é quase sempre muito menor que o número de linhas no conjunto de treino T, a complexidade de tempo é significativamente menor e por isso ela foi escolhida.

3 Experimentos e Resultados

Os experimentos abaixo foram executados repetidas vezes variando os parâmetros designados por cada seção abaixo. Esses testes foram executados em um computador com processador Intel Core I7 de 3.60GHz e 8 núcleos de processamento, com 16GB de memória DDR4 e sistema operacional Ubuntu 18.04.1 LTS.

Vale ressaltar que, como os testes teriam que ser variados, a semente para as funções *random* foi desativada, a fim de conseguir alcançar diferentes resultados para cada execução do algoritmo. Para executar o *script* interativo de execução de testes, basta executar os comandos abaixo:

```
cd src/
chmod +x tests.sh
./tests.sh
# Você será perguntado de quantas repetições do teste deseja executar
```

Todos os testes abaixo foram executados com os valores padrões, com exceção do parâmetro que está sendo variado, conforme descrito abaixo:

```
POP=100
GEN=20
MUT=0.05
CROSS=0.9
K=2
SEED=0
ELIT=1
ELITOPS=1
```

e foram realizados na base sintética *Synth2*.

3.1 Tamanho da População

O tamanho da população variou no conjunto {20, 100, 200, 500, 1000}. Conforme o número aumentou, o tempo de execução do algoritmo ficou muito maior, porém a solução encontrada no final foi relativamente melhor ($fitnessBest_{pop=100} \leq fitnessBest_{pop=500}$), conforme pode ser observado pelo gráfico abaixo:

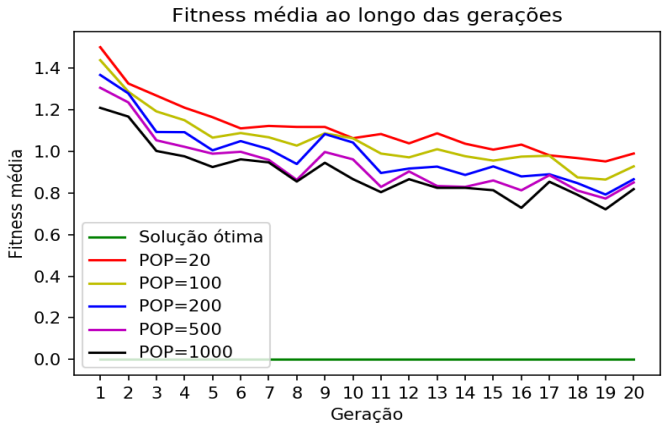

Figura 7 – Fitness média ao longo das gerações com a variação do tamanho da população

No entanto, essa melhoria foi pequena, quando comparado aos outros parâmetros que foram variados.

3.2 Número de Gerações

O número de gerações variou no conjunto {20, 40, 60, 80, 100}. Conforme o número aumentou, o tempo de execução do algoritmo ficou muito maior (inclusive do que aumentou a população), porém a solução encontrada no final foi relativamente melhor ($fitnessBest_{gen=20} \leq fitnessBest_{gen=100}$), conforme pode ser observado pelo gráfico abaixo:

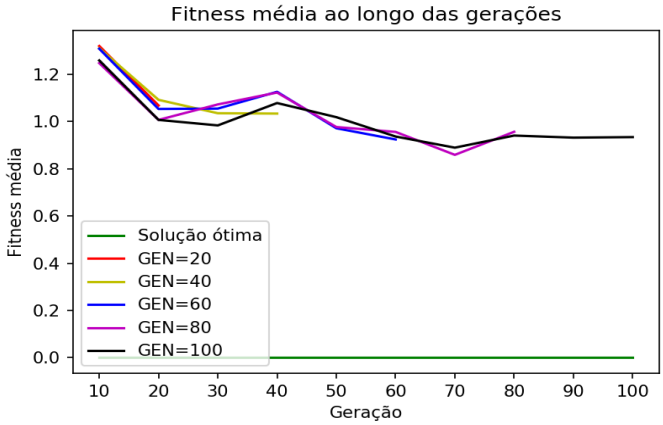

Figura 8 – Fitness média ao longo das gerações com a variação do número total de gerações

No entanto, essa melhoria também foi pequena, quando comparado aos outros parâmetros que foram variados.

3.3 Mutação e Cruzamento

A probabilidade de mutação e cruzamento variou no conjunto $\{(0.05, 0.9), (0.2, , 0.7), (0.4, 0.6), (0.6, 0.4), (0.9, 0.05)\}$, respectivamente. Isso possibilitou que, quando a probabilidade de mutação aumentou e a probabilidade de cruzamento diminuiu, foi possível observar que indivíduos sofreram maiores alterações aleatórias, o que trouxe consigo uma enorme diferença quanto à *fitness* dos indivíduos.

Algumas vezes os indivíduos gerados eram significativamente piores que os indivíduos do primeiro teste, mas em outras vezes eles eram significativamente melhores. Isso pode ser observado abaixo nos gráficos:

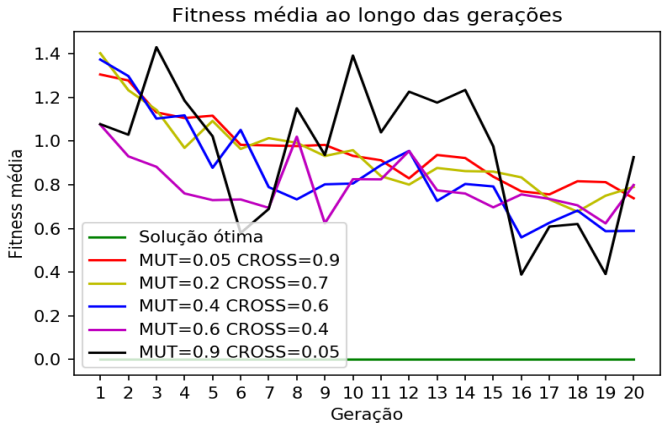

Figura 9 – Fitness média ao longo das gerações com a variação de $P_{mutação}$ e $P_{cruzamento}$

Portanto, a variação desse parâmetro gera um comportamento na população que é facilmente observado. Desse modo, o impacto da variação da probabilidade de mutação e probabilidade de cruzamento se mostrou significativo.

3.4 Tamanho do Torneio

O tamanho do torneio variou no conjunto $\{1, 2, 4, 10, 50\}$. Conforme o torneio aumentou de tamanho, a velocidade de convergência da população aumentou significativamente:

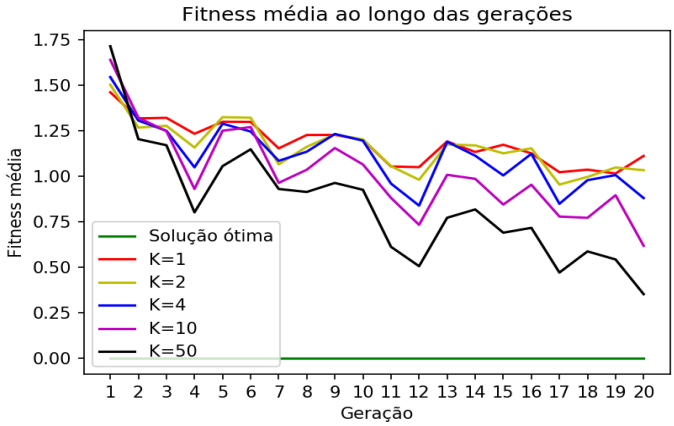

Figura 10 – Fitness média ao longo das gerações com a variação do tamanho do torneio

Portanto,o tamanho do torneio se mostrou um importante parâmetro, conforme ele foi variado. Para valores maiores de K_α, a pressão seletiva foi muito grande, fazendo com que a velocidade de convergência fosse significativamente maior que para um determinado K_β, sendo $K_\alpha > K_\beta$.

3.5 Operadores Elitistas

A ativação e desativação de operadores elitistas causou um enorme impacto na velocidade de convergência e no espaço de busca das soluções, conforme pode ser observado no gráfico abaixo:

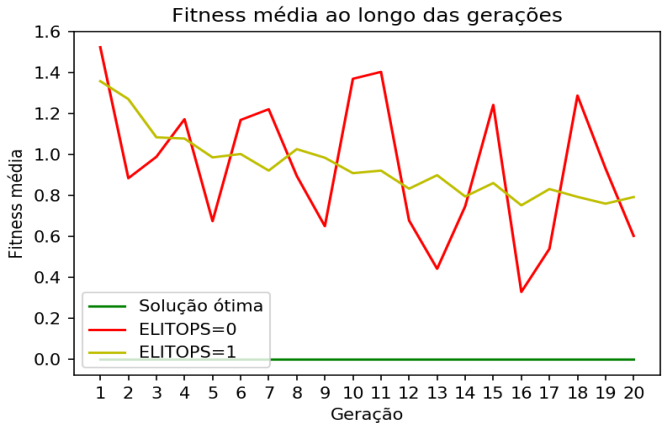

Figura 11 – Fitness média ao longo das gerações com a ativação/desativação de operadores elitistas

Portanto, a desativação de operadores elitistas geraram uma maior diversidade nas populações, de modo que um filho sempre substitua seu pai em operadores genéticos, resultando em uma velocidade de convergência mais lenta. Já a ativação desses operadores geraram uma menor diversidade nas populações, de modo que um pai que seja muito bom (*fitness* pequena) nunca seja substituído em futuras gerações, resultando em uma velocidade de convergência significativamente mais rápida.

3.6 Elitismo

A ativação e desativação de elitismo não se mostraram muito impactantes nas execuções dos testes realizados. Isso se dá principalmente ao fato de que, se um pai é relativamente melhor que seus filhos gerados ao longo das diversas gerações de uma execução do algoritmo, ele será sempre colocado na próxima geração por causa dos operadores elitistas.

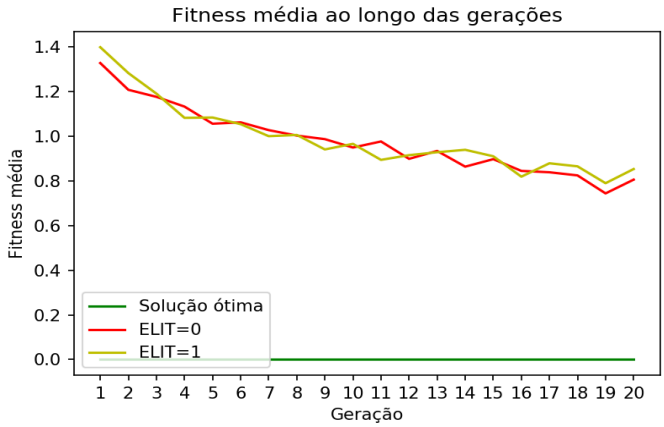

Figura 12 – Fitness média ao longo das gerações com a ativação/desativação de elitismo

Quando foram realizados testes desativando os operadores elitistas, não foi possível realizar conclusões concretas das execuções dos algoritmos, uma vez que o desvio padrão da *fitness* média das soluções estava muito alto, impossibilitando uma análise mais concreta em tempo hábil, pois seria necessária a execução de muitas iterações de testes para tal.

3.7 Bloating

Para a medida de *bloating*, seria necessário a comparação de indivíduos pelos seus resultados gerados a partir do conjunto de teste dado como entrada. Essa estratégia, mesmo que muito custosa, detectaria indivíduos que possuem íntrons em suas subárvores, como por exemplo, um indivíduo expresso pela função matemática $(x_0)^2$ é igual aos indivíduos $x_0 * x_0$, $(x_0)^1 * (x_0)^1$, $(\sqrt{x_0})^2 * (\sqrt{x_0})^2$, entre outros diversos tipos de variações dessas funções. No entanto, esse é um exemplo simples e fácil de detectar, mas indivíduos mais complexos podem tornar isso totalmente não-trivial, mesmo que utilize de algoritmos de redução.

3.8 Execução Synth1

Os parâmetros que se mostraram mais efetivos para a aproximação da solução ótima foram:

```
POP=100
GEN=50
MUT=0.4
CROSS=0.6
K=2
ELIT=1
ELITOPS=0
```

e, portanto, foram utilizados para execução na base de dados Synth1. Para esses testes, foi possível observar que a maior parte das soluções finais, que possuíam menor *fitness* no conjunto de treino, obtiveram um aproveitamento similar no conjunto de teste, levando a acreditar que esses parâmetros realmente foram bem escolhidos.

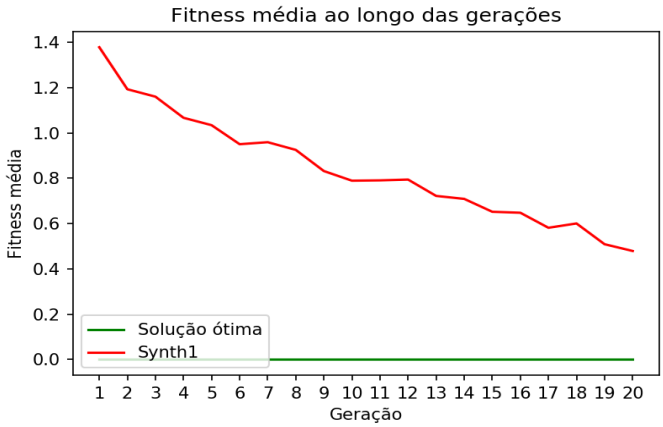

Figura 13 – Fitness média ao longo das gerações com os parâmetros otimizados encontrados

3.9 Execução Concrete

Para os testes no conjunto de dados real foram variados dois parâmetros que se mostraram mais impactantes nos testes na base de dados sintética *Synth2*. No entanto, os parâmetros que se mostraram mais impactantes anteriormente não foram tão impactantes comparativamente na base de dados real. No entanto, foram gerados testes para comparação da variação de tais parâmetros abaixo.

3.9.1 Operadores Elitistas

Os operadores elitistas foram ativados e desativados nos testes executados. Porém, os resultados não foram tão significativos quanto os testes no Synth2:

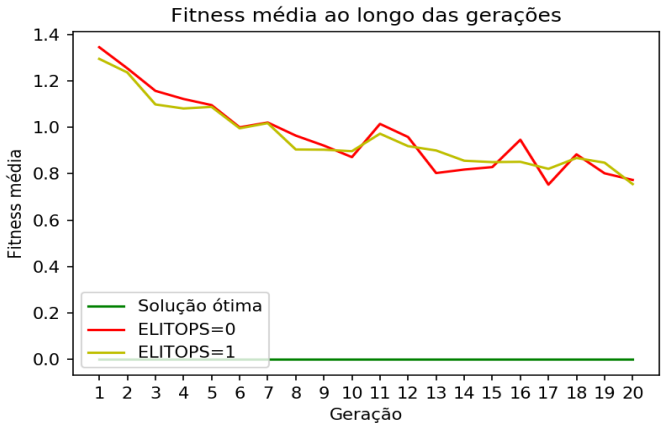

Figura 14 – Fitness média ao longo das gerações com a ativação/desativação de operadores elitistas

Portanto, podemos concluir que mesmo passando ou não o melhor pai para a próxima geração, o espaço de busca se manteve quase sempre restrito, provocando um comportamento similar tanto na ativação quanto na desativação de operadores elitistas.

3.9.2 Tamanho do Torneio

O tamanho do torneio foi variado no conjunto $\{1, 2, 4, 10, 50\}$. Nesse caso, os resultados foram bastante significativos quanto à velocidade de convergência da solução final:

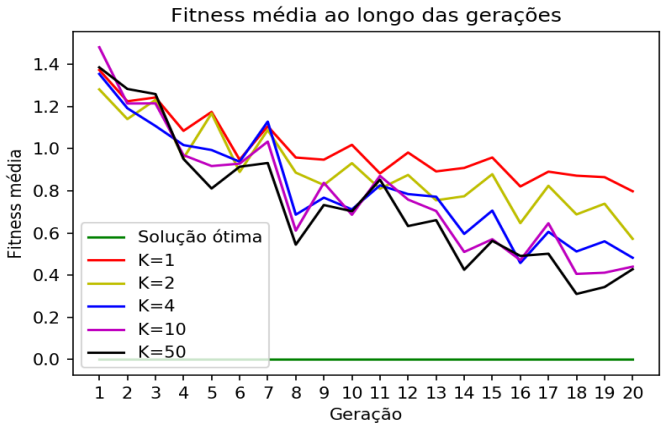

Figura 15 – Fitness média ao longo das gerações com a variação do tamanho do torneio

Portanto, podemos concluir que a variação do parâmetro K exerce uma enorme influência na velocidade de convergência em todos os testes executados nesse trabalho. Isso se dá pela forte relação do K com a pressão seletiva em algoritmos de Programação Genética.

4 Conclusões

Esse trabalho prático foi muito agregador no sentido de exercitar e procurar entender o quanto algumas variações de parâmetros impactam na solução final do algoritmo. Ele ajudou a fixar alguns conceitos de Programação Genética, bem como geração aleatória da população inicial, como representar um indivíduo, como gerar uma nova população a partir de mutações, cruzamentos e reproduções da população anterior, como alterar a pressão evolutiva com o tamanho do torneio, entre outros diversos tópicos importantíssimos para a área de Computação Natural e para algoritmos probabilísticos como um todo.

Para tal, a confecção de testes variando esses parâmetros se mostrou uma tarefa exaustiva e demorada (pela repetições e tempo de finalização da execução de alguns testes), porém totalmente necessária para o entendimento do assunto proposto. No entanto, uma combinação exata desses parâmetros, de forma a prover uma solução ótima ou até mesmo aproximada à ótima, ainda se mostrou de uma complexidade enorme, mas foi possível determinar certos subconjuntos de combinações que melhoravam relativamente essas execuções em busca dessas soluções.

Embora em alguns casos a melhor solução alcançada tenha sido somente uma constante em que a sua *fitness* foi a mínima, em outros foi possível encontrar árvores que expressam funções totalmente não-triviais que se aproximavam mais ainda da solução ótima. Talvez o fato de que o poder computacional ao meu dispor para a realização desses testes é pequeno em relação ao tempo que era necessário para a execução completa do

algoritmo me privou de encontrar soluções mais complexas, porém mais aproximadas do ótimo no primeiro caso citado.

Para a resolução do problema de regressão logística, a programação genética quase sempre tem um ótimo desempenho quando comparado à outros demais métodos, incluindo várias técnicas de aprendizagem de máquina, pela solução[4]. Isso demonstra o tamanho poder de algoritmos de Computação Natural para a resolução de problemas complexos em que são necessários diversos cálculos e estratégias distintas de comparação de resultados.

5 Referências Bibliográficas

[1] WIKIPEDIA. Symbolic regression. https://en.wikipedia.org/wiki/Symbolic_regression. Acessado em 29 de Setembro de 2018.

[2] WIKIPEDIA. Randomized algorithm. https://en.wikipedia.org/wiki/Randomized_algorithm. Acessado em 29 de Setembro de 2018.

[3] WALKER, M. Introduction to genetic programming. *Tech. Np: University of Montana*, 2001.

[4] ORZECHOWSKI, P.; LA CAVA, W.; MOORE, J. H. Where are we now? a large benchmark study of recent symbolic regression methods. *arXiv preprint arXiv:1804.09331*, 2018.