Trabalho Prático 1 - Regressão simbólica via Programação Genética

André Luiz Moreira Dutra - 2019006345
Computação Natural

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

cienciandre@ufmg.br

1. Introdução

O problema da regressão simbólica é um problema de regressão no qual se deseja aproximar o comportamento de uma função de n parâmetros por meio do aninhamento de diferentes funções, aplicadas a combinações dos parâmetros possíveis e constantes. Este trabalho se propõe a buscar uma solução para este problema por meio de um modelo baseado em programação genética, bem como analisar diferenças e impactos nos resultados obtidos quando diferentes hiperparâmetros do modelo são utilizados.

2. Modelagem do problema e detalhes de implementação

Nesta seção será descrito como cada elemento da PG foi modelado no sistema, bem como decisões teóricas e detalhes práticos sobre a implementação de cada elemento serão especificadas. Dentre esses elementos se incluem, dentre outros, a geração da população inicial, a definição dos operadores genéticos, escolhas quanto à implementação dos métodos de seleção e a modelagem do indivíduo.

2.1. Modularização

Para a implementação do problema, a modelagem por programação genética foi interpretado como um subconjunto dos algoritmos evolucionários, variando apenas na modelagem do organismo, que no caso da PG é feita usando árvores. Assim, o programa foi modularizado nos seguintes pacotes:

EA_framework: Implementa um algoritmo evolucionário genérico. Oferece interfaces para a implementação do método de geração de indivíduos, do método de seleção e da implementação do organismo nas classes Population_generator, Selector e Organism. O algoritmo em si está implementado na classe Evolutionary_algorithm, que utiliza elementos de cada interface para gerar a nova população a cada iteração.

organism_implementations: Contém as implementações relacionadas à modelagem do organismo. A classe Srgp_organism implementa Organism no contexto de GPs para regressão simbólica. E as demais classes implementam diferentes formas de gerar a população inicial e implementam Population_generator.

selector_implementations: Contém as implementações dos diferentes métodos de seleção. As classes desse pacote implementam a interface Selector.

symbolic_regression_tree: Contém a implementação da árvore de símbolos. Contém a classe abstrata do nó, classes que implementam o nó para nós não-terminais e terminais e funções de geração de árvores. Mais detalhes sobre a implementação da árvore serão descritos em próximos tópicos.

2.2. Modelagem do indivíduo

O indivíduo foi modelado na classe Srgp_organism, utilizando árvores para o genótipo e funções lambda para o fenótipo. As árvores modelam o genótipo na forma de nós, cuja implementação básica foi feita na classe abstrata Sr_node, de maneira que funções e operadores nos tenham sido implementados como nós não-terminais e variáveis e constantes como terminais, ambos os casos herdando de Sr_node. Os nós concretos foram implementados nas classes Binary_function, Unary_function, Variable e Constant.

A classe Binary_function contém funções e operadores de dois parâmetros, que correspondem a não-terminais com dois filhos na árvore, e foram escolhidos como opções os operadores +, -, * e ÷.

A classe Unary_function contém funções que só aceitam um parâmetro, e correspondem a não-terminais com um filho na árvore, tendo como única função representante sin (a função seno).

A classe Variable contém as variáveis da função, que correspondem a terminais na árvore.

Por fim, a classe Constant corresponde a constantes, que também são terminais. As constantes da solução foram geradas como um valor aleatório entre 0 e 5.

O fenótipo, por sua vez, foi implementado utilizando funções lambda. Dado que cada nó pode ser associado a uma função que recebe como entrada a lista de parâmetros e retorna uma função que os combina, junto a cada nó foi guardada uma

função lambda que recebia a lista de variáveis e retornava a aplicação da função ou operador do nó ao resultado das funções lambda de cada filho aplicadas às mesmas variáveis. No caso de terminais, a lambda retorna ou a constante ou a variável associada. Assim, o resultado é uma função lambda que recursivamente se comporta como a árvore a descreve.

Funções utilitárias gerais aos nós foram implementadas na classe Sr_node, como funções para imprimir o conteúdo, ou atualizar os filhos. Funções específicas de cada nó foram implementadas nas respectivas classes concretas, como a geração das funções de retorno. Por fim, funções de geração de árvores aleatórias com múltiplos nós foram implementadas no módulo Sr_tree.

Além das implementações do genótipo e fenótipo a partir das árvores, o Srgp_organism também implementa os operadores genéticos e a fitness. Foi optado por fazer dessa forma pois, nesse caso, os operadores genéticos e a fitness não dependem de outros indivíduos que não o alvo, então a separação deles na implementação do indivíduo tornaria a arquitetura mais modularizada. Além disso, calculando a fitness no indivíduo, é possível utilizar ele próprio para guardar a fitness para caso ela precise ser calculada novamente, o que é especialmente relevante considerando que, para quase todos os métodos de seleção, o cálculo da fitness depende do número de exemplos dos dados de treino. Assim, a fitness, após calculada pela primeira vez, é guardada em uma variável interna do organismo se for indicado com uma flag que a fitness deve ser salva. Assim, se uma próxima chamada à fitness salva for feita, o valor da variável interna é retornado. Para o cálculo da fitness, cada organismo recebe uma referência ao dataframe com os dados de treino, para que a fitness possa ser calculada a partir dele.

2.3. Definição dos operadores genéticos

Conforme mencionado, os operadores genéticos foram implementados dentro da classe Srgp_organism, que representa o organismo da PG. No caso da mutação, um nó aleatório é escolhido na árvore, com probabilidades iguais para todos exceto a raiz, que nunca é escolhida, e este nó é substituído por uma subárvore aleatória. A árvore aleatória é gerada utilizando o método grow e tem altura máxima igual a metade da altura da árvore pai. Já o cruzamento é feito escolhendo um nó aleatório de cada indivíduo, com probabilidades iguais para todos exceto a raiz, e trocando as subárvores.

Conforme mencionado, as raízes nunca são alteradas, pois a implementação foi favorecida dessa forma. Assim, em árvores com mais de um nó os pais nunca são colocados como opções de escolha, e em árvores com apenas um nó os operadores genéticoss simplesmente não são aplicados, retornando uma cópia da árvore original. no entanto, como árvores de apenas um nó são pouco comuns na geração de indivíduo, isso não é um problema grande.

2.4. Geração da População Inicial

A geração da população inicial foi implementada de três formas, em três diferentes classes.

A classe Grow_population_generator implementa o gerador de populações com árvores aleatórias pelo método grow, que escolhe a partir do nó raiz nós filhos entre todos os possíveis nós para montar a árvore, terminais ou não-terminais, gerando árvores menos uniformes e mais diversas. Apesar disso, o nó raiz foi implementado para sempre ser não-terminal, para evitar o problema do tópico anterior.

A classe Full_population_generator implementa o gerador de populações com árvores aleatórias pelo método full, que gera árvores completas e balanceadas para a altura máxima dada, escolhendo apenas não-terminais em todas as camadas até a penúltima, escolhendo terminais para a última. De forma semelhante, o nó raiz é sempre um não-terminal.

A classe Rhh_population_generator implementa o gerador de populações com árvores aleatórias pelo método ramped half-and-half, que gera populações iguais de árvores grow e full em intervalos de altura da mínima à máxima até que a população total seja formada.

A intenção do trabalho era avaliar também as diferenças entre os métodos de geração de população, mas devido ao extenso tempo de cada execução foi optado por manter a geração por ramped half-and-half em todos os experimentos.

2.5. Métodos de Seleção

Os métodos de seleção solicitados foram implementados no pacote selector_implementations em três classes diferentes.

A classe Roulette_selector implementa a seleção por roleta. A implementação foi feita de forma análoga à disposição das fitness de cada indivíduo em uma fita

contínua: ao escolher um ponto aleatório da fita, cada indivíduo teria probabilidade proporcional à fitness de conter o ponto. Assim, a implementação foi feita fazendo uma lista de fitnesses cumulativas e escolhendo um número aleatório entre 0 e a soma total, definindo como vencedora a primeira fitness cumulativa a conter o número. Para que as fitnesses menores tivessem mais chance, já que nesse caso a fitness é o RMSE, que deve ser minimizado, a roleta foi aplicada para o inverso das fitnesses de cada indivíduo.

A classe Tournament_selector implementa a seleção por torneio da população, recebendo k como parâmetro. Como no torneio basta que a melhor seja escolhida, as fitness não foram tomadas como inversas, e sim como seu valor original e escolhendo como melhor o indivíduo de menor fitness.

A classe Lexicase_selector implementa a seleção pelo método lexicase apresentado. É notável que, como o seletor e a fitness estão em classes diferentes, o lexicase traria problemas na implementação modularizada, já que requeriria mudar os dados do organismo. No entanto, isso foi contornado permitindo no organismo o cálculo da fitness com apenas um subíndice dos dados, que é passado como parâmetro da função. Assim, o lexicase foi feito calculando as fitness de cada indivíduo passando como subíndice apenas a linha avaliada em cada iteração. Como se trata de um problema de regressão, sem fitness discreta, foi implementado o epsilon-lexicase. Porém, pelas mesmas razões das diferenças de geração de população, o tempo de execução dos algoritmos fez com que se optasse por um epsilon fixo de 0.3, ou seja, apenas organismos com fitness até 0.3 maior que o melhor eram escolhidos para a próxima seleção. Mas esse valor poderia facilmente ser adaptado para ser passado como parâmetro na implementação.

2.6. Looping de geração da nova população

O Looping de geração da nova população foi a única classe implementada no EA_framework, na classe Evolutionary_algorithm. Ele basicamente, dada uma instância de organismo, gerador de população e seletor, aplica o ciclo padrão de todo algoritmo evolucionário: Gera a população inicial, calcula a fitness, realiza a seleção de um par de indivíduos, aplica mutação ou cruzamento com determinadas probabilidades e adiciona os indivíduos à nova população, até que o tamanho desejado da população seja atingido. Por fim, determina se o critério de parada foi atingido e, se sim, calcula as fitnesses e retorna o melhor indivíduo e, se não, reinicia o ciclo.

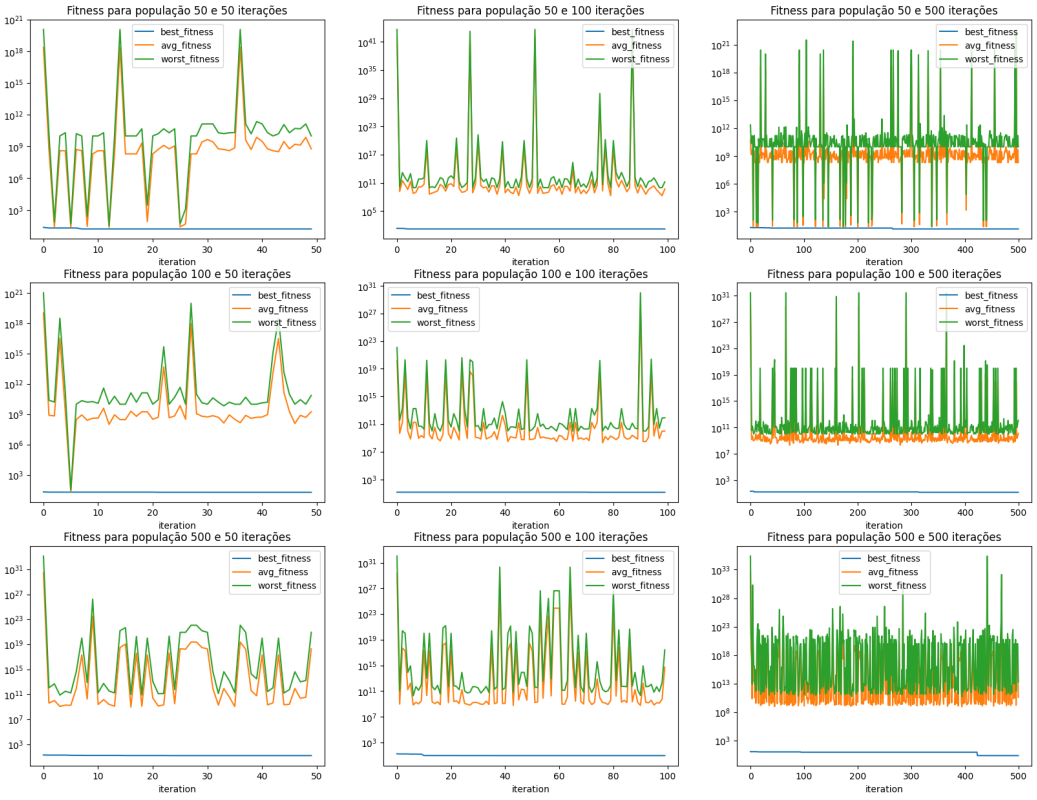
Para a implementação do looping, foi definido que o critério de parada seria apenas o número de iterações, já que nenhum limiar de confiança foi definido para as soluções do problema. Além disso, o método guarda as estatísticas de cada iteração do algoritmo em um dataframe e o salva como csv. Esse arquivo está definido como statistics na implementação da main, enquanto o arquivo results contém os resultados do algoritmo para a base de testes.

3. Análise dos Resultados

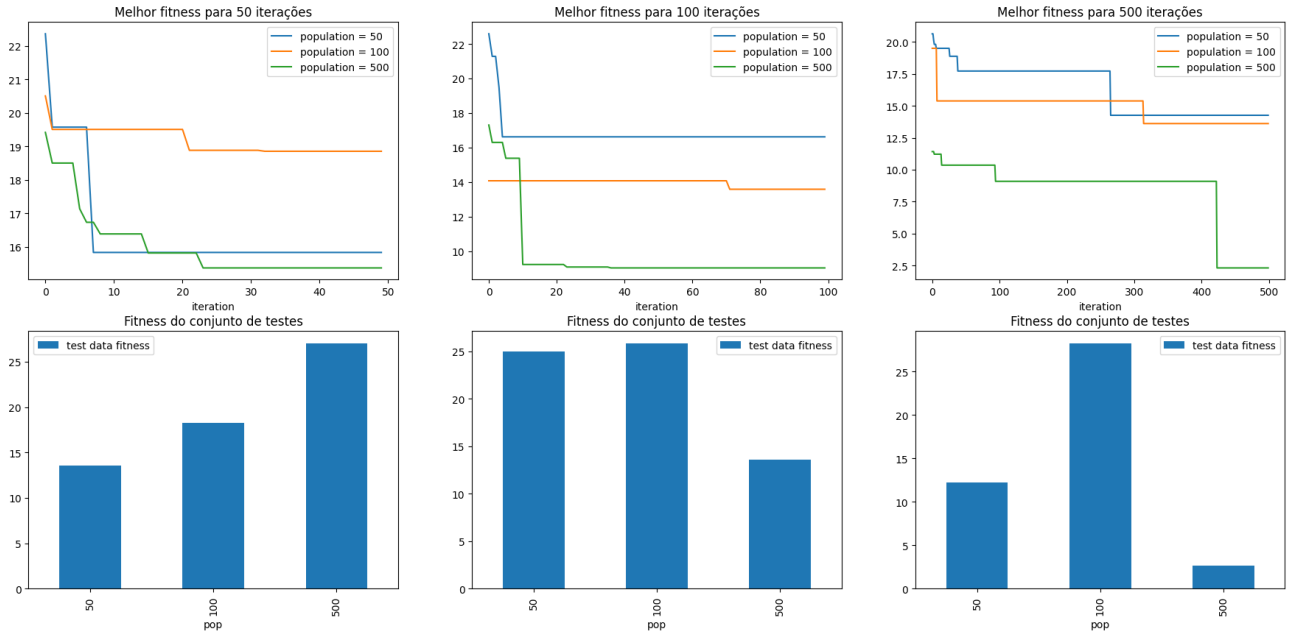
Nesta seção, o algoritmo será aplicado aos diferentes conjuntos de dados sugeridos, de maneira que possamos observar diferenças nos comportamentos para cada conjunto de parâmetros escolhido. Dentre os conjuntos de dados, estão synth1 e synth2, que representam dados sintéticos (gerados com alguma combinação simbólica), e concrete, que representam dados concretos, extraídos de uma base real.

3.1. Synth1

Primeiramente, variamos a população e o número de iterações entre os valores 50, 100 e 500, com seletor em torneio com k=2, mutação p_m=0.05 e cruzamento p_c=0.9:

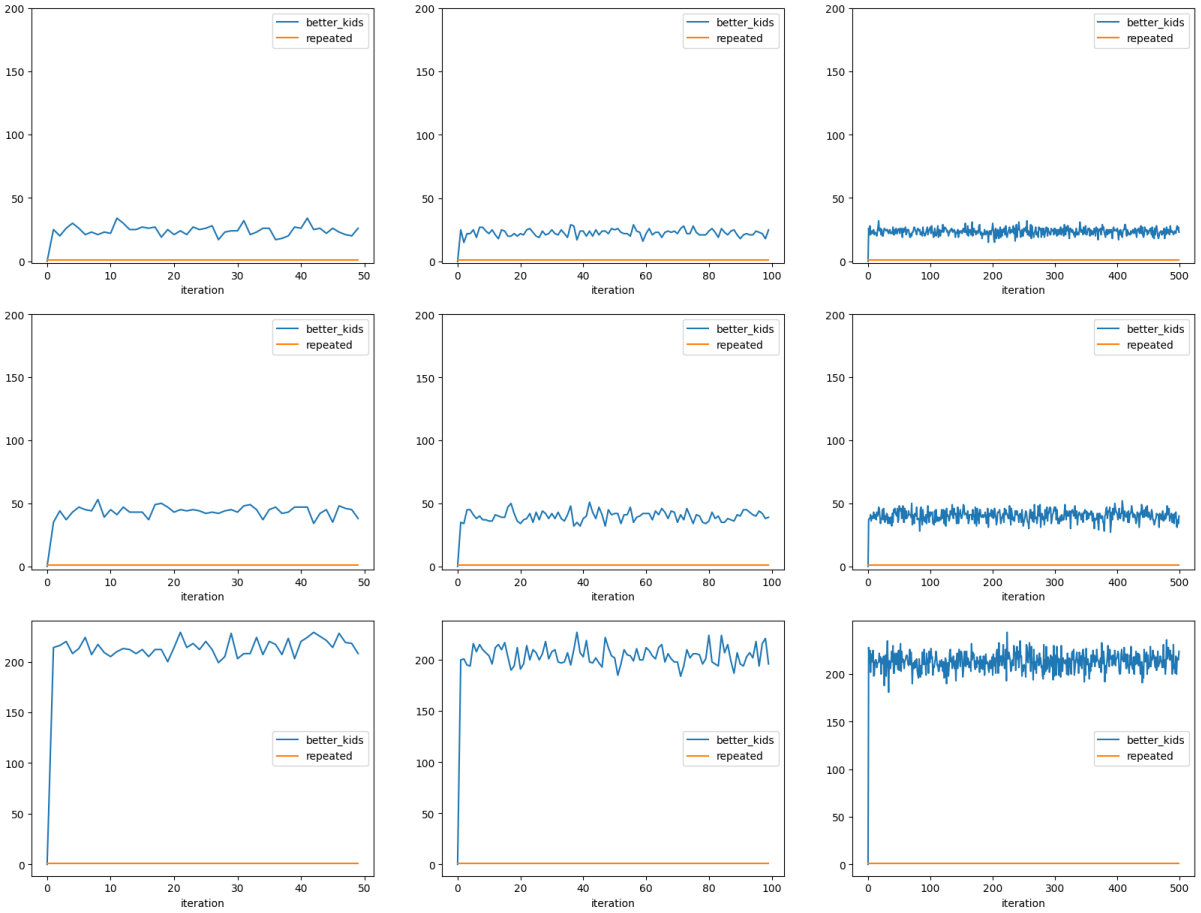


A característica mais evidente do gráfico é a diferença entre as melhores soluções e as soluções médias, que estão bem mais próximas das piores. Note que a escala da fitness é logarítmica e mesmo assim a solução está ordens de grandeza abaixo desde o início. Contraintuitivamente, o comportamento da fitness média e da pior se mantêm em um fluxo aleatório, diferente do que se esperaria, que seria a diminuição gradual de ambas à medida que mais iterações ocorrem e a população é selecionada. Mas esse comportamento é compreensivo, uma vez que k=2 oferece o mínimo de pressão seletiva que o método de seleção por torneio pode oferecer. Analisemos apenas as melhores fitness, junto às fitness resultantes do teste:

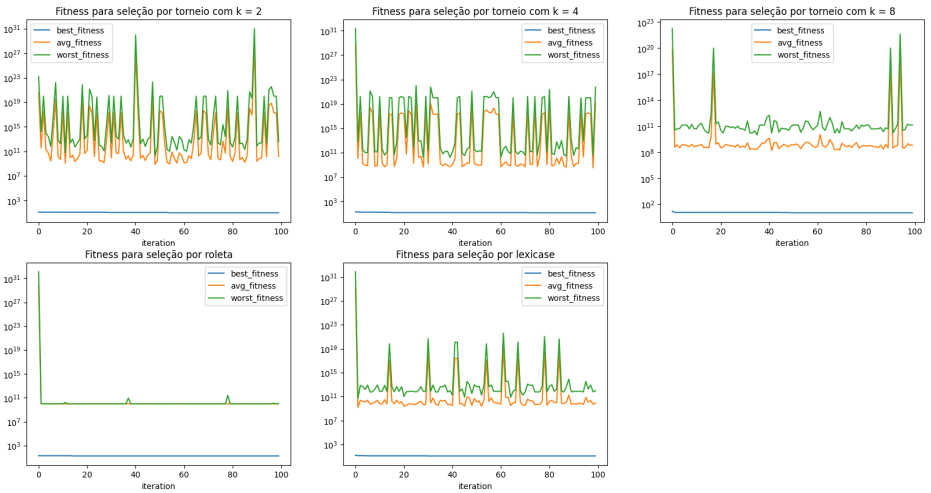


Pelo comportamento dos algoritmos em cada iteração fica evidente uma tendência de que maiores populações convergem a fitness menores, mas o primeiro caso surpreendentemente apresenta um desempenho pior da maior população no teste. Isso muito provavelmente se dá porque, por se tratar do menor número de iterações, possivelmente o algoritmo ainda não tinha tido iterações o suficiente para convergir.

Tendo estes resultados em mente, foi definido que uma maior população e maior número de iterações seriam o ideal. No entanto, a máquina em que o experimento foi executado não foi capaz de executar os algoritmos para estes parâmetros, então ficou definido o tamanho da população como 500, com 100 iterações cada



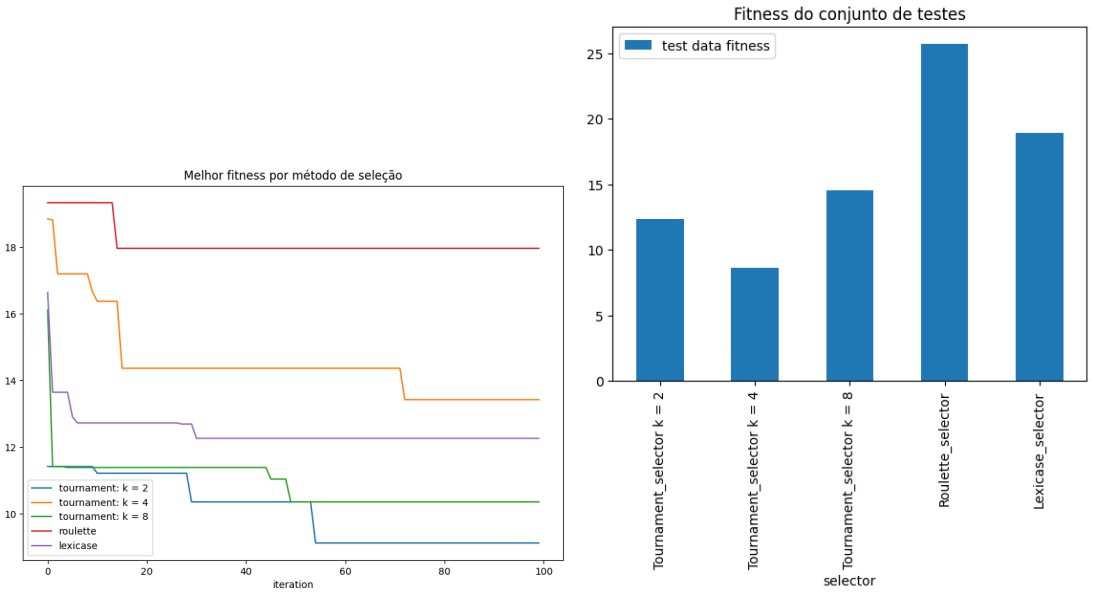
Acima temos o plot de quantos indivíduos superaram a média dos seus pais em cada iteração, bem como a plotagem do número de indivíduos repetidos em cada população. Possivelmente o cálculo de repetidos sofreu algum bug, já que todos os algoritmos apresentam exatamente uma repetição, mas o número de filhos melhores que os pais reafirma a observação sobre o comportamento aleatório do algoritmo neste estágio, com a exceção de que o intervalo onde esse número rodeia é proporcional ao tamanho da população escolhida, o que também é esperado.



Acima temos as fitness de diferentes modelos testados para o problema fixando os parâmetros de população e número de iterações encontrado antes e variando o método de seleção. Foram testados cinco métodos de seleção, sendo eles três por torneio com k variando entre 2, 4 e 8, um por roleta e um por epsilon-lexicase, com epsilon fixo em 0.3.

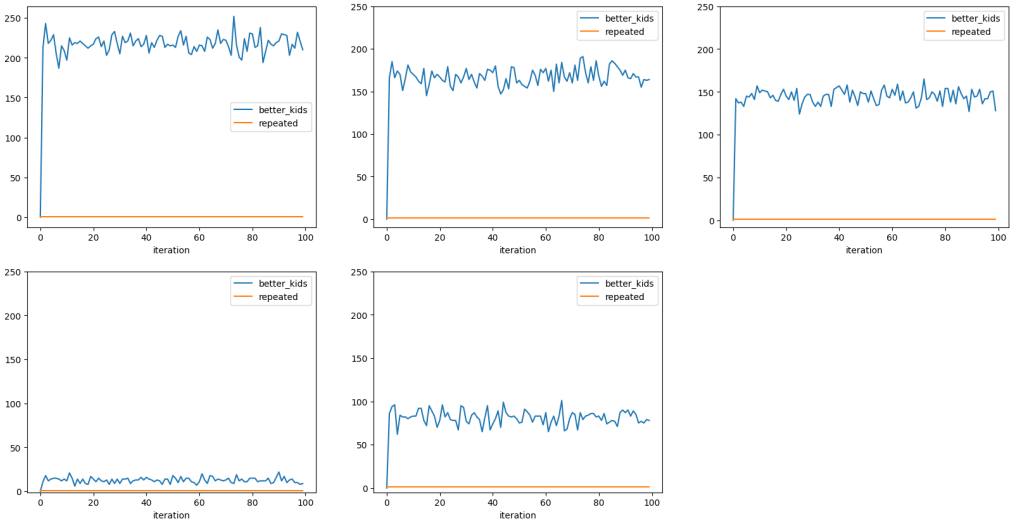
De maneira semelhante, uma queda na fitness média e na pior ao longo das iterações não é vista, o que é particularmente inesperado. Nesse caso o comportamento é ainda mais incomum, uma vez que métodos de seleção com maior pressão seletiva foram testados dessa vez e mesmo assim em nenhum há uma tendência de descida. Isso pode ser visto na seleção por roleta e na seleção por torneio com k=8, por exemplo, onde a variabilidade é bem menor mas mesmo assim não há tendência de descida.

Observando a melhor fitness do treino e a fitness do teste, temos:



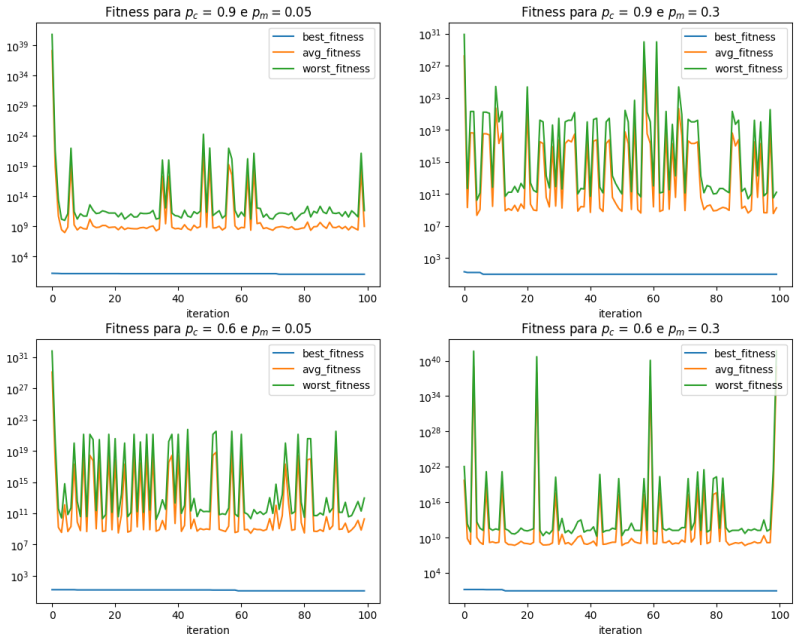
Em ambos os casos o melhor método de seleção foi o torneio com k = 4, então ele foi escolhido para continuar com os testes.

Abaixo, temos o plot de quantos filhos foram melhores que a média dos pais:

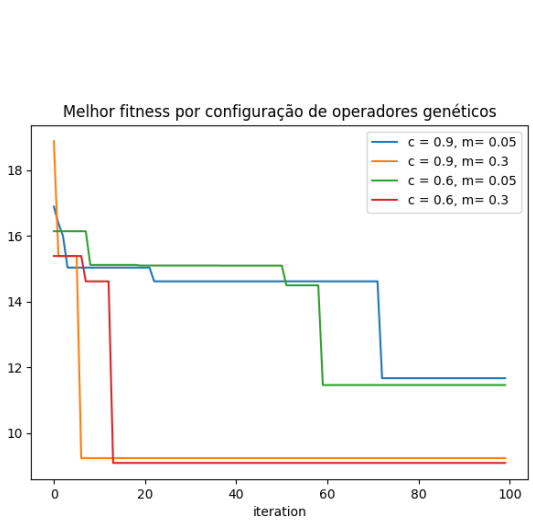


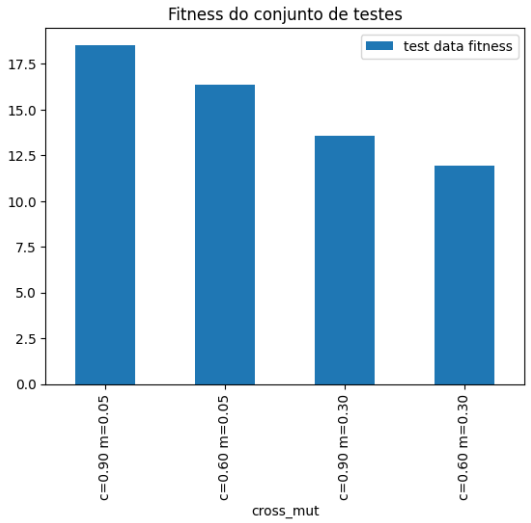
Note como as fitness que se apresentavam mais uniformemente no gráfico com todas as fitnesses desse conjunto apresentam menos filhos menores que os pais neste, na média. De fato, ambas as características estão ligadas à pressão seletiva do modelo. Em um modelo com alta pressão seletiva, os melhores têm grande chance de serem escolhidos localmente, então a probabilidade de a mutação ou crossover piorar uma boa fitness selecionada anteriormente é grande, o que pode ser observado neste gráfico. Por se tratar de um método elitista, como a chance de os pais serem melhores que os filhos é grande, a chance do melhor da geração anterior ser melhor que o da atual também é grande, o que explica a uniformidade observada no primeiro gráfico.

Selecionando por torneio com k = 4, variamos as probabilidades de mutação e crossover, conforme abaixo:



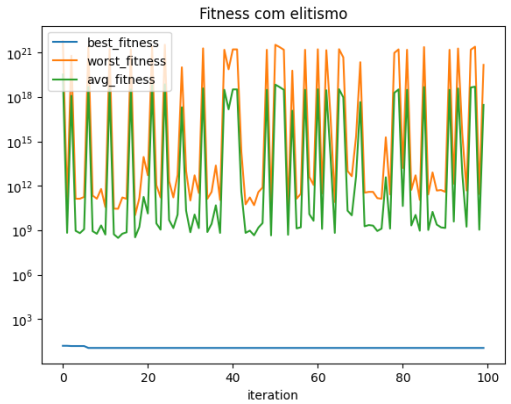
O algoritmo segue não apresentando melhoria gradual ao longo das iterações. Observemos os melhores resultados do treino e do teste:

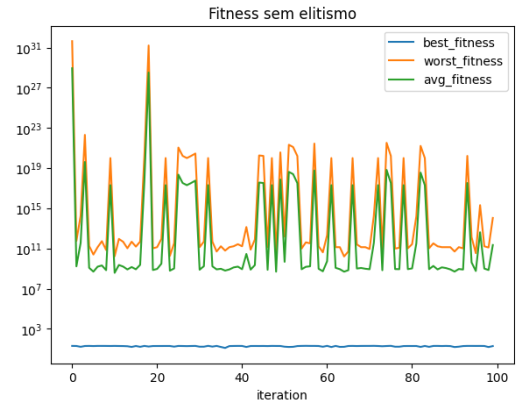




Embora uma tendência muito clara não seja visível entre as fitness médias, é evidente que, tanto para o treino quanto para o teste, um crossover baixo e uma mutação alta tiveram melhor desempenho. Isso é especialmente interessante e esperado considerando que, no caso de GPs, muitas vezes o cruzamento tem um efeito destrutivo na informação, uma vez que não troca pedaços das árvores necessariamente semanticamente equivalentes.

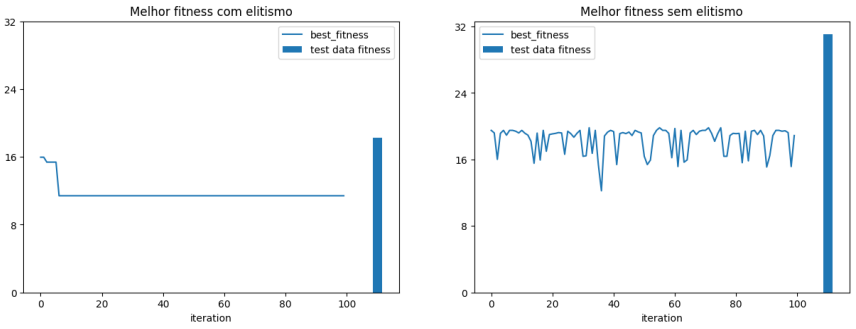
Assim, selecionando p_c = 0.6 e p_m = 0.3, partimos para a análise com relação ao uso ou não de elitismo no modelo:





Mesmo o elitismo não alterou a falta de tendência de descida do modelo, então muito provavelmente isso está relacionado a algum detalhe da implementação. Possivelmente, o método de seleção usado, ramped half-and-half, é bom em gerar populações iniciais diversas, já que algoritmos com fitness baixa se apresentam na população desde o início. Também é

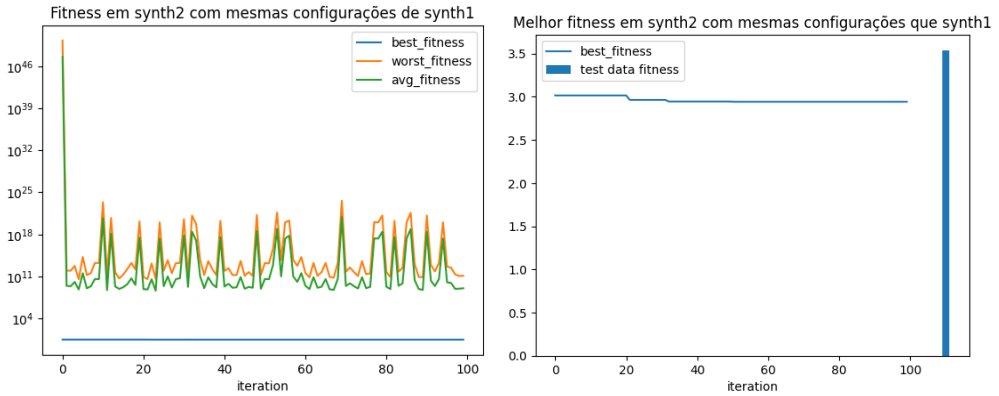
possível que os métodos de seleção não estejam aplicando pressão suficiente, ou que algum bug no algoritmo esteja desconsiderando a implementação feita nos seletores.



Por essa imagem é evidente, no entanto, como os algoritmos elitistas nesse caso têm um comportamento melhor, tanto no conjunto treino quanto no teste

3.2. Synth2

Para synth2, foram aplicadas as mesmas configurações tidas como ótimas em synth1, e as fitness foram calculadas:

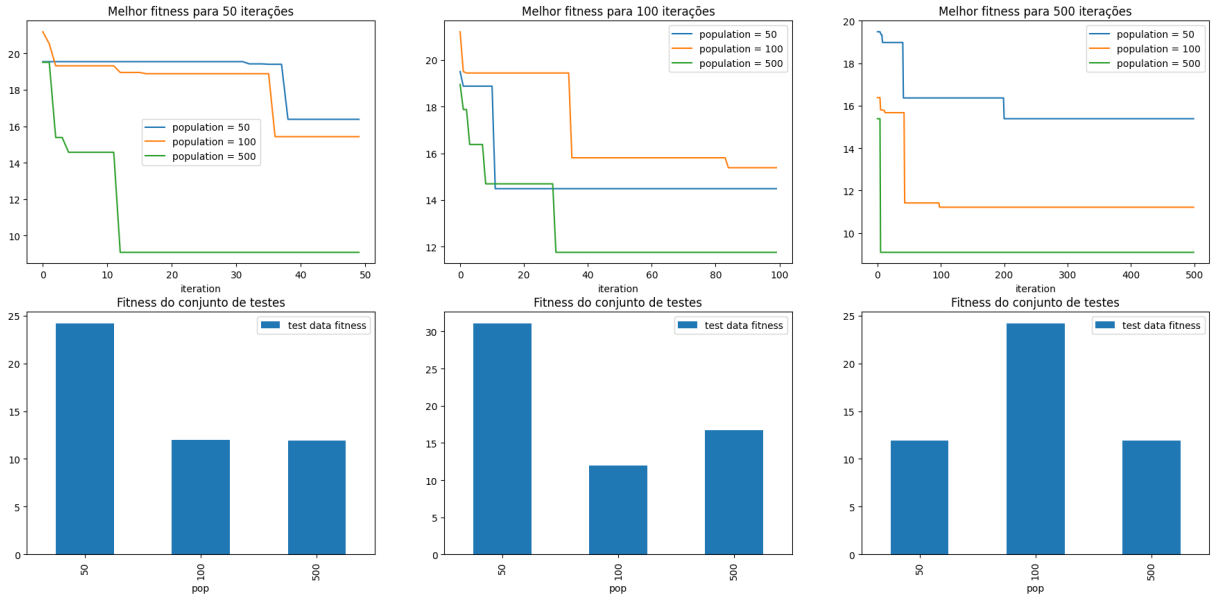


O resultado obtido em synth2 foi excepcionalmente bom, mesmo considerando o conjunto teste. Embora a média tenha se mantido alta, os melhores indivíduos chegaram a fitnesses de menos de 5, o que, no contexto do RMSE de um banco inteiro, é consideravelmente bom.

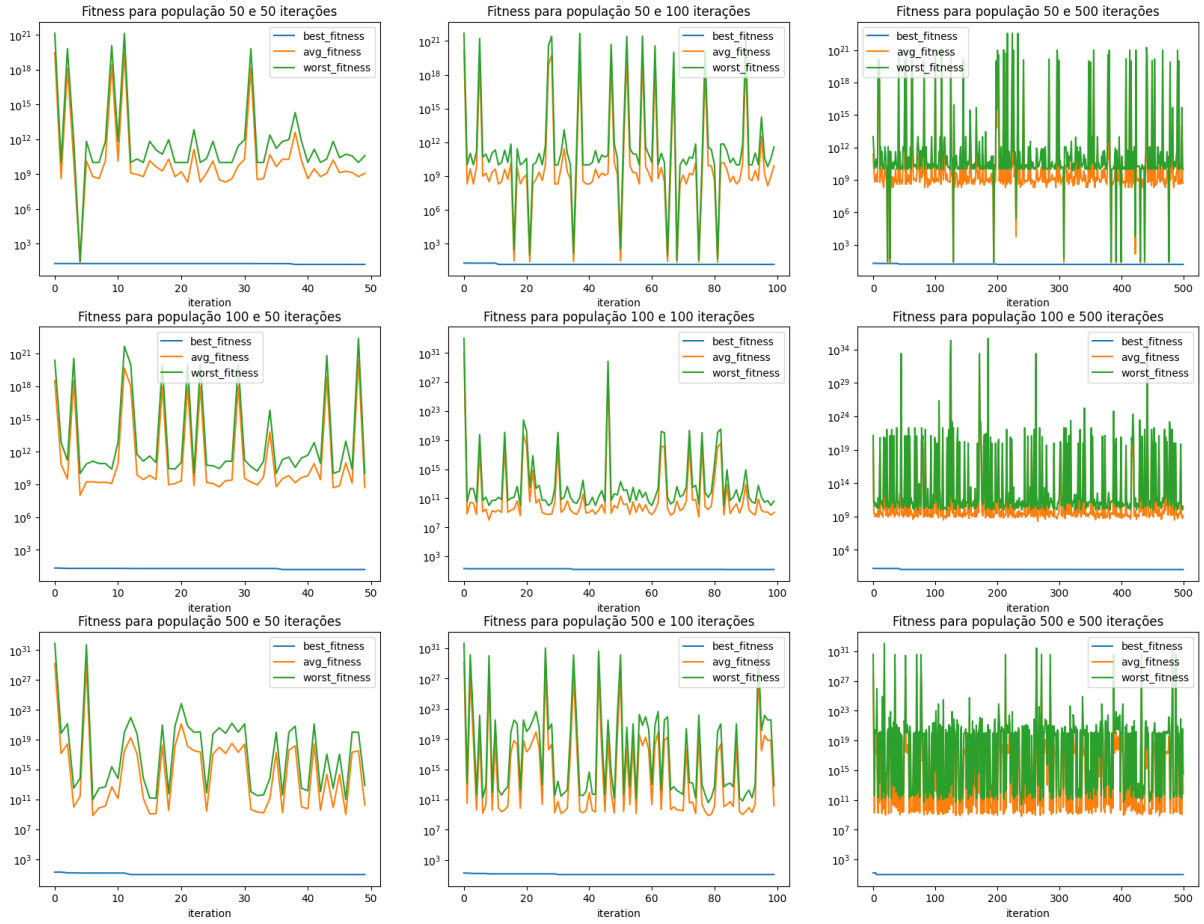
3.3. Concrete

A base concrete foi extraída de uma base de dados real, e consiste em uma função de 7 parâmetros (sendo o oitavo o resultado). Faremos o mesmo processo que fizemos com synth1 para testar o limite de capacidade da GP com essa base.

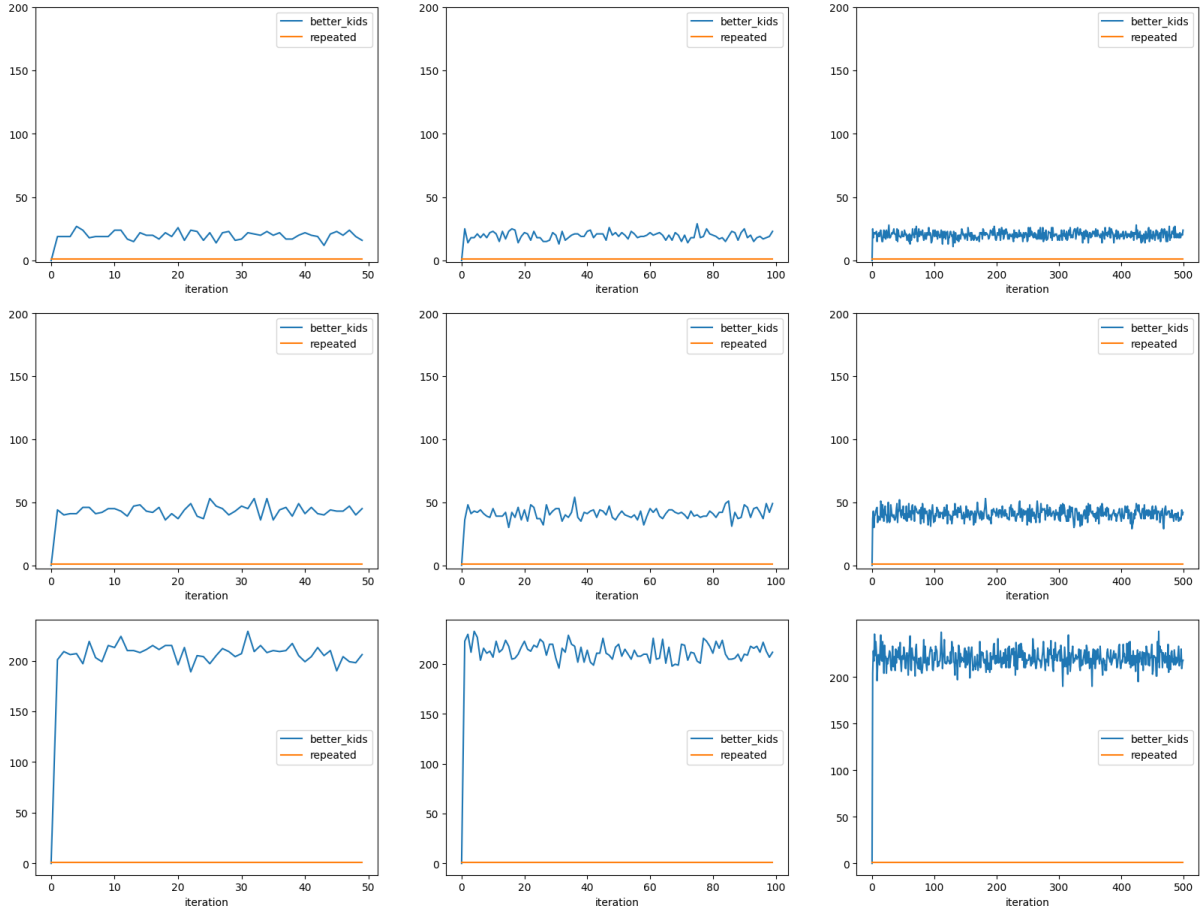
Primeiro, vamos variar a população e o número de iterações:



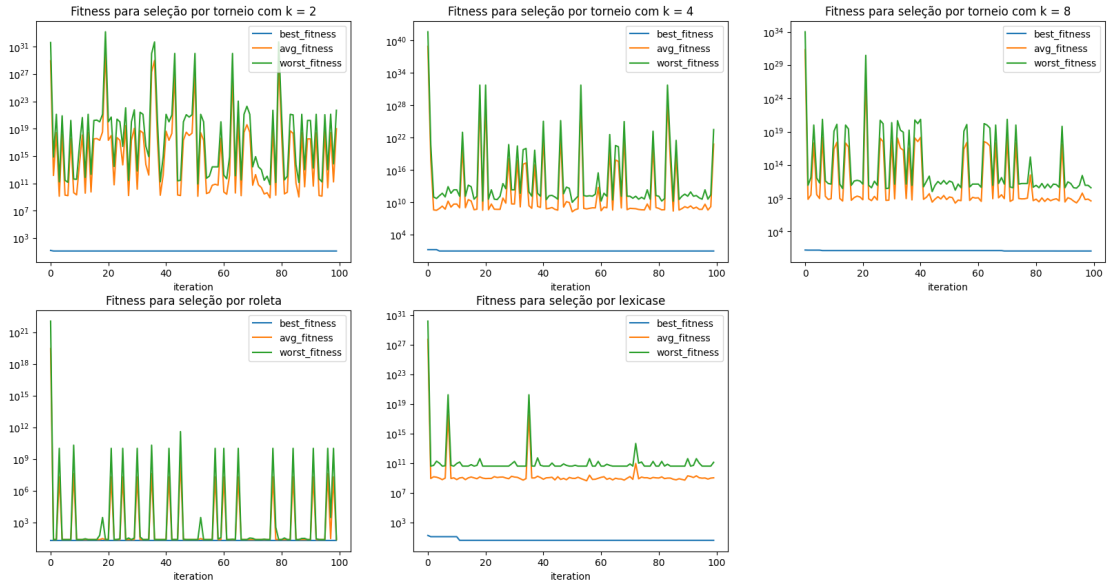
De forma semelhante à anterior, maiores populações com maiores iterações trouxeram melhores resultados. Ao invés de 500 iterações, selecionamos 100 para acelerar a execução, mas a população se manteve com 500 indivíduos.



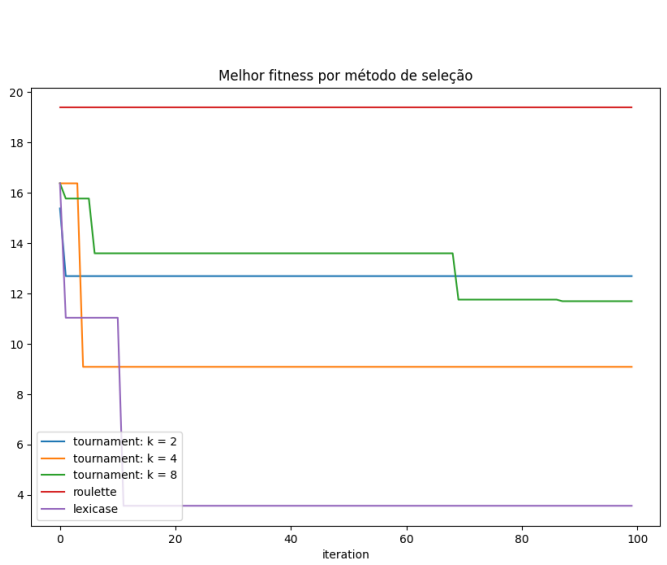
Com relação às fitness médias, o mesmo comportamento pseudoaleatório pode ser visto. O mesmo pode se dizer com relação à quantidade de filhos melhores que os pais proporcional ao tamanho da população utilizada, conforme visto abaixo:

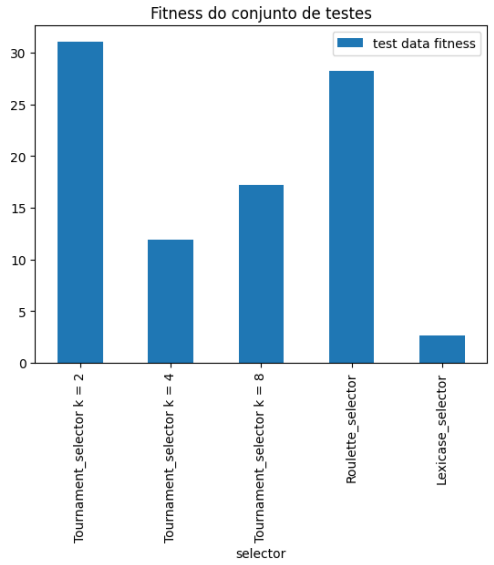


Com população de 500 indivíduos e 100 iterações, avançamos para a análise dos seletores:

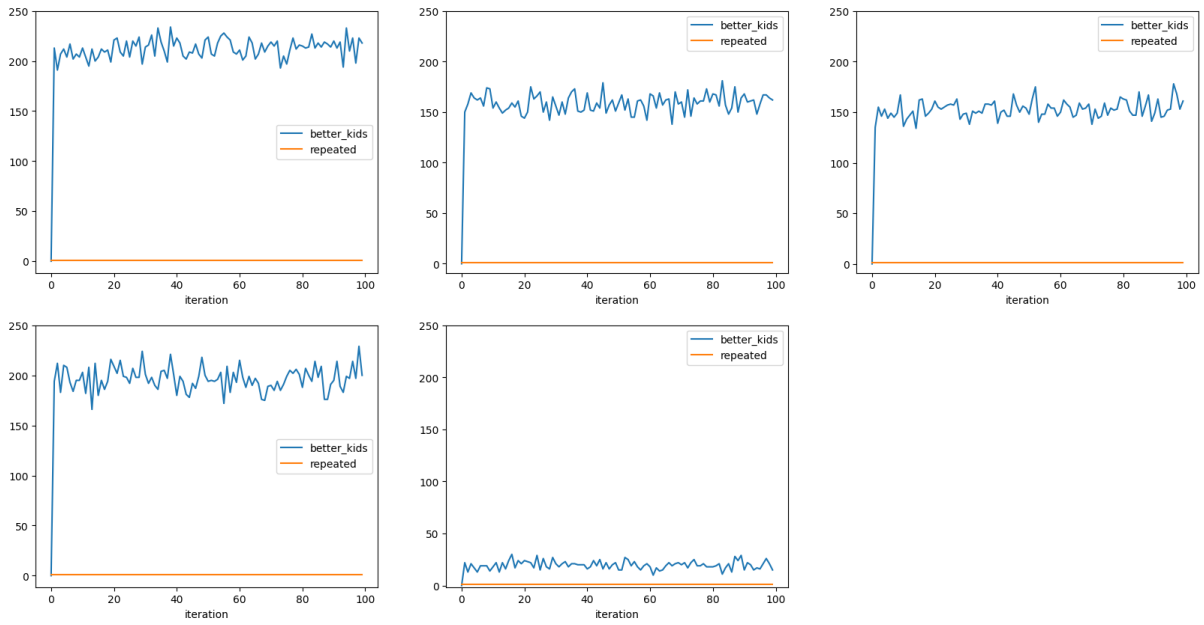


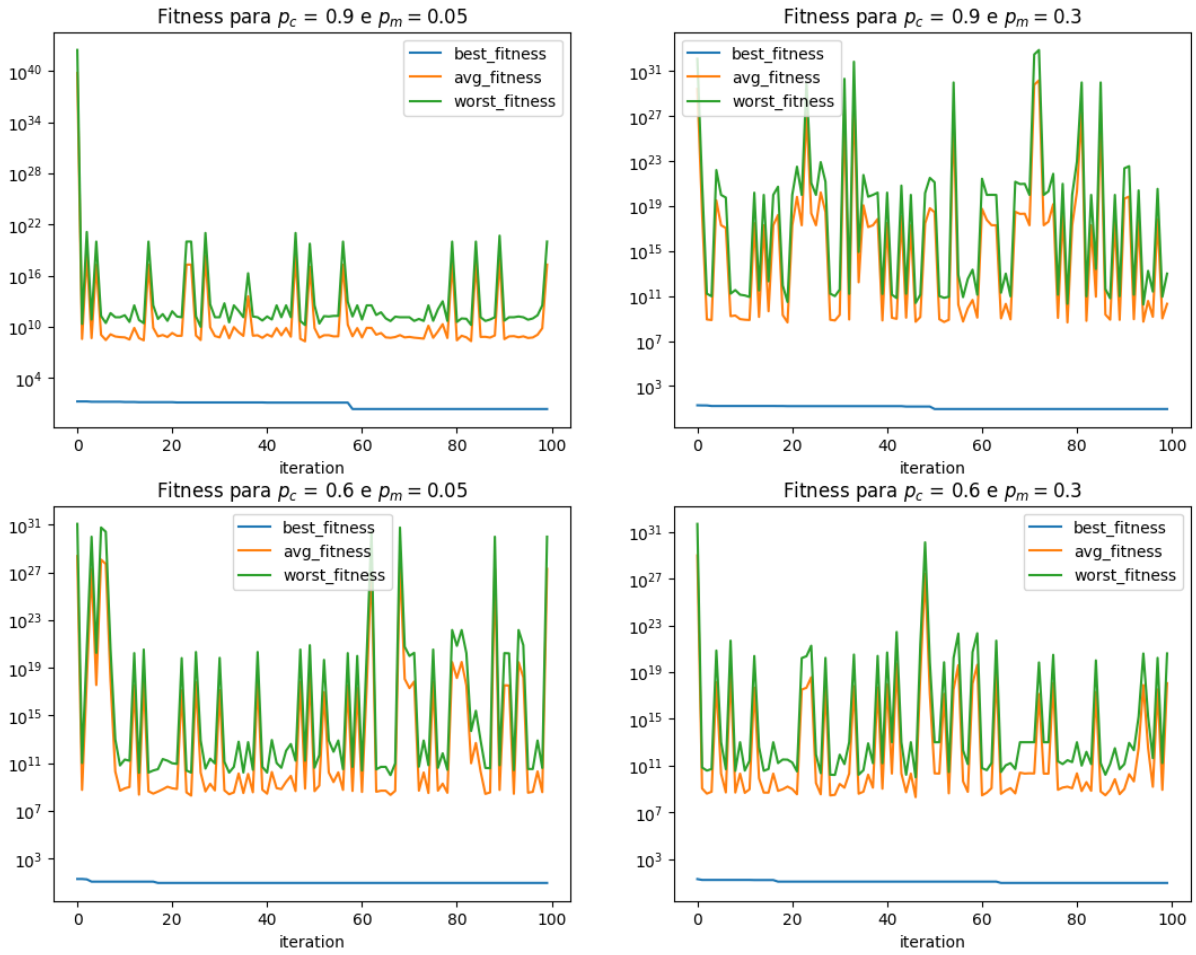
Nesse caso, é notável que o lexicase e o roleta se mantiveram bem mais uniformes que o restante, com o lexicase variando ainda menos que o roleta. Isso pode indicar uma maior pressão seletiva, ou uma tendência do problema real que favoreça ou não diferentes métodos de seleção.



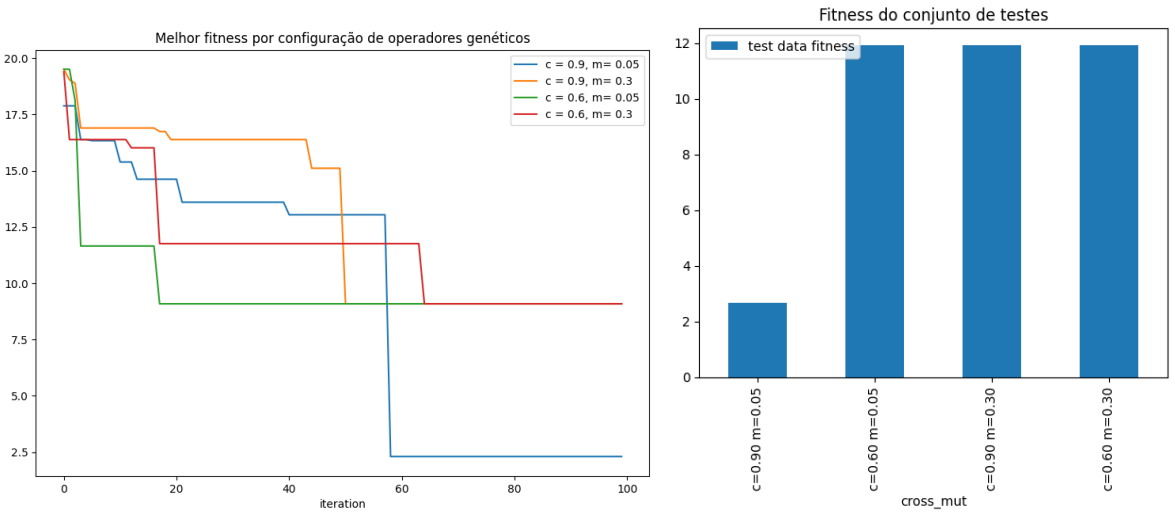


Em ambos os casos o lexicase foi bem melhor que os demais métodos, tornando-o o seletor escolhido para as próximas etapas. Observando abaixo, podemos ver que o lexicase tem muito menos filhos melhores que os pais que todos os outros métodos, indicando um comportamento de alta seletividade evolutiva neste contexto.

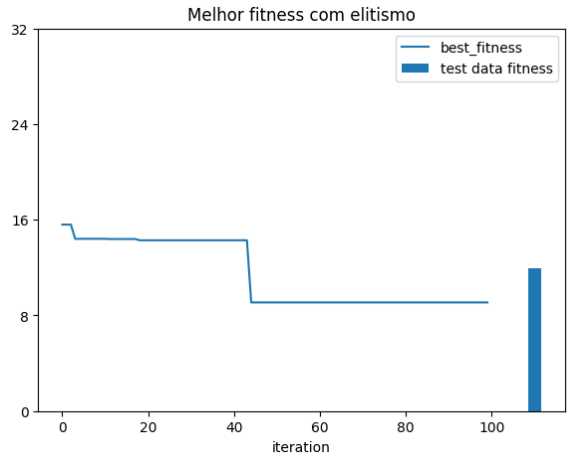


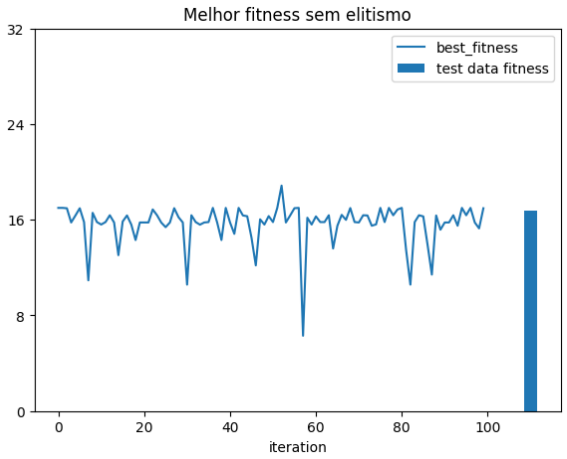


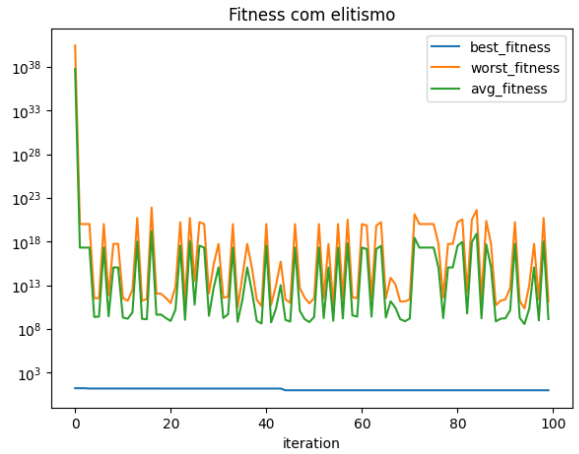
Avaliando as diferenças entre as configurações da p_c e p_m, curiosamente nesse caso a configuração de maior crossover e menor mutação é mais uniforme que as demais. Ela também apresenta melhor desempenho no treino e no teste, conforme pode ser visto:

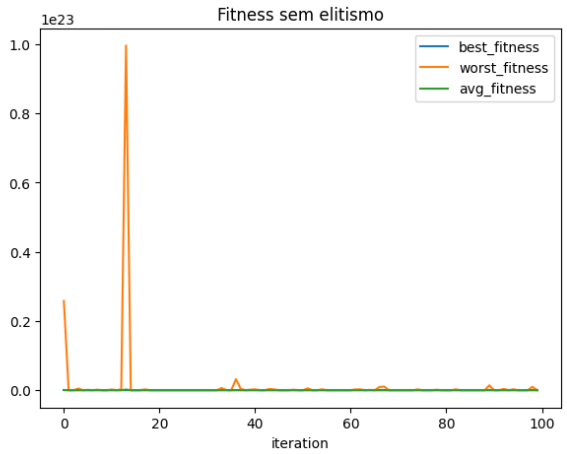


Embora contraintuitivo, já que não se espera um bom desempenho com muito crossover, esse comportamento foi utilizado para a escolha da p_c e p_m do algoritmo.









Por fim, basta observar as diferenças entre o uso de elitismo ou não. Com relação aos resultados da melhor fitness, ele se comportou como os demais, sendo mais uniforme com o uso de elitismo e menos uniforme sem o uso, mas ainda orbitando um valor próximo do mínimo elitista encontrado. No entanto, a média e a pior fitness no organismo sem elitismo aparentemente convergiram muito rápido, já que são os mais próximos da solução real até então, com poucas variações. Isso é indicativo de um método com muita pressão seletiva, o que deve ser corrigido aumentando o epsilon do lexicase.

4. Conclusão

Este trabalho permitiu a observação na prática do comportamento de EAs e PGs, bem como testar exaustivamente seus possíveis parâmetros. Uma das principais conclusões tiradas foi a de que muito provavelmente há algum erro na implementação dos seletores, já que em nenhum caso houve uma tendência de descida da fitness com a evolução do algoritmo. Também foi interessante observar a influência da pressão seletiva dos algoritmos em diferentes estatísticas dos organismos, mesmo que ela não tenha sido quantificada para alguns dos algoritmos em que foi observada, como o da roleta e o lexicase.

Bibliografia

PAPPA, Gisele. *Slides da disciplina Computação Natural. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2023.*

pandas.DataFrame.plot - pandas 0.22.0 documentation. Disponível em: <https://pandas.pydata.org/pandas-docs/version/0.22/generated/pandas.DataFrame.plot.html> . Acesso em: 23 maio. 2023.