Trabalho Prático 1 - Programação Genética

Júlia Fonseca de Sena - 2018054508

1. Introdução

O trabalho foi implementado com o intuito de desenvolver e aplicar conceitos da programação genética. Para isso, utilizou-se um algoritmo com regressão simbólica para encontrar a melhor função de distância para determinar a classificação de linhas de uma base de dados. A representação das funções escolhida foi por árvore, na qual cada nó representa uma função ou um terminal, sendo o terminal um parâmetro de um dos dois pontos cuja distância está sendo calculada.

A linguagem utilizada foi Python3, e as bibliotecas foram: pandas e numpy para manipulação de dados; copy para fazer *deep copies* que podem ser alteradas sem afetar o original; random para gerar números aleatórios; sklearn para a medida da fitness e implementação dos clusters hierárquicos; time para medir os tempos médios das gerações e do algoritmo geral; e gc para liberar memória do que não será mais utilizada.

2. Implementação

2.1 Representação

A representação escolhida foi a árvore binária, onde cada nó aponta para um nó à sua direita e um à sua esquerda e um valor (string) que é uma função ou um terminal. As funções cobertas pelo algoritmo são as básicas (+, -, x, ÷) e os terminais são duas vezes o número de parâmetros da base (um parâmetro de cada ponto dos dois que a distância será calculada). Ou seja, se a base de dados possui 3 parâmetros, os terminais são P10, P11, P12, P20, P21, P22, nos quais os dois primeiros caracteres identificam qual o ponto, e do terceiro em diante, o parâmetro.

A fim de haver diversidade nas populações, embora toda a população inicial tenha altura 7, as árvores não são completas. Dessa forma, quando ocorre cruzamento, podem surgir árvores menores que uma de altura 7. A construção ocorre colocando funções até que um dos ramos alcance a altura 6, depois preenchendo os nós faltantes com os terminais (toda função deve ter um terminal ou outra função nos dois nós a que se liga: na esquerda e na direita).

A computação da função representada é feita a partir da raiz, indo para os outros nós por recursão. Quando encontra um terminal, retorna o valor que representa (para os dois pontos passados para a função cuja distância está sendo calculada). Quando é uma função, recebe os valores do ramo à esquerda e à direita, e realiza a operação de acordo com o valor do nó: se a string do nó for ‘+’, soma, ‘-’, subtrai, etc.

Para o clustering hierárquico ao invés de por k-means (que foi sugerido como mais rápido no fórum e aprovado pela professora), criou-se uma função que recebe o *dataframe* sem a classificação correta e manda pares de pontos para a função de distância. A função *AgglomerativeClustering* não faz isso automaticamente, foi preciso utilizar também a função *pairwise_distances*, ambas da sklearn.

2.2 Operadores genéticos

Os operadores utilizados foram crossover, mutação, elitismo e seleção por torneio. O crossover foi feito escolhendo um ramo aleatório de cada um dos dois indivíduos recebidos até que seja possível trocá-los sem que um dos filhos tenha altura maior que 7. Para a mutação, se o nó do ramo aleatório escolhido for uma função e a alteração for para

outra função, basta mudar a string do nó para a nova operação, mas se for para um terminal também é preciso tornar os nós da esquerda e da direita *None*, já que terminais são sempre folhas, não possuem filhos. Já se for passar um terminal para função, é um pouco mais complicado, já que é preciso criar também um ramo aleatório. Para isso, foi criado uma função que popula o ramo até que não haja funções com ramos vazios ou chegue à altura máxima, no caso 7.

No elitismo, é criado um *dataframe* com a população e a fitness, o qual é ordenado decrescentemente pela fitness, e é retornado o indivíduo que está em primeiro lugar. Na seleção por torneio, k indivíduos aleatórios e não repetidos são escolhidos, e aquele com maior fitness é escolhido e tem seu *index* nos *arrays* de população e fitness retornado.

2.3 Fitness

A fitness é calculada após o *clustering* dos dados utilizando a função de distância do indivíduo em questão. O *clustering* é feito com os comandos da sklearn supracitados, e a fitness do indivíduo é calculada com a medida V, comparando o cluster em que os dados foram colocados, baseado na solução que representa, e sua categoria real. O resultado é colocado num *array* no mesmo índice que o indivíduo ocupa no *array* da população.

2.4 Geração

Em cada geração, é calculado a fitness média, máxima e mínima, quantos indivíduos são iguais, quantos filhos gerados no crossover são melhores e quantos são piores que a fitness média de seus pais. Essas estatísticas de cada geração, juntamente com a fitness da melhor solução encontrada com a base de dados para testes, foram utilizados para encontrar a melhor configuração de parâmetros (número da população, número de gerações, probabilidades de crossover e de mutação e se o elitismo é aplicado ou não) na metodologia experimental.

A fim de criar uma nova população por geração, primeiro checa-se se há elitismo (se houver, coloca-se o melhor indivíduo da geração anterior na nova. Seguindo, até que a população seja do tamanho desejado, seleciona-se um indivíduo por torneio, e verifica se ele fará crossover ou mutação, de acordo com as probabilidades. Os filhos são gerados primeiramente como *deep copies* dos pais, e então são enviados para a respectiva função do operador selecionado, a fim de manter os pais sem alteração para operações seguintes.

Para achar a estatística de filhos melhores e piores que os pais, a função que cria uma nova população também retorna um vetor que possui a média das fitness dos pais que geraram o indivíduo ou -1 (no caso de ter sido criado por elitismo ou mutação).

A função de geração é repetida pelo número de gerações definidos, e as estatísticas de cada geração são salvas em um *dataframe* para análise posterior. Após o fim das gerações, é retornado o indivíduo com maior fitness da última população, e, com a base de dados de teste, calcula-se novamente sua fitness.

3. Experimentos

A avaliação experimental foi feita com a base de câncer de mama (breast_cancer_coimbra), sendo o modelo treinado por breast_cancer_coimbra_train.csv e testado por breast_cancer_coimbra_test.csv. Para avaliar a melhor configuração de parâmetros, testou-se valores diferentes para os parâmetros, variando um de cada vez para avaliar os efeitos.

Primeiro, variou-se o tamanho da população:

- 30 indivíduos, a média da fitness do conjunto de teste para os experimentos foi 0,07174, o desvio padrão foi 0,06037, o tempo médio por geração foi cerca 40 segundos, a melhor fitness para o treino foi 0,24333 e para o teste 0,12563;
- 50, a média foi 0,17885 e o tempo por geração foi 67 segundos, desvio padrão foi 0,07424, a melhor fitness no treino foi 0,175013 e do teste foi 0,23136;
- 100, média de 0,14274 e 180s por geração, desvio padrão foi 0,07508, a melhor fitness no treino foi 0,19937 e do teste foi 0,18072;
- 500 chegou a uma média de 732 segundos por geração e uma fitness média de 0,18568, desvio padrão foi 0,06208, a melhor fitness no treino foi 0,20865 e do teste foi 0,19881.

Observando os tempos e a melhoria das fitness do treino e teste com o aumento da população, foi concluído que o tamanho que possui maior custo-benefício é 50.

A próxima análise a ser feita é quanto ao número de gerações. Já ajustando a população para 50, os dados obtidos foram os seguintes:

- 30 gerações, a média da fitness do conjunto de teste para os experimentos foi 0,15129, o desvio padrão foi 0,06975, o tempo médio por geração foi cerca 68 segundos, a melhor fitness para o treino foi 0,23163 e para o teste 0,17871;
- 50, a média foi 0,15045 e o tempo por geração foi 92 segundos, desvio padrão foi 0,03510, a melhor fitness no treino foi 0,11915 e do teste foi 0.17527;
- 100, média de 0,0759315 e 173 segundos por geração, desvio padrão foi 0.00828, a melhor fitness no treino foi 0,186164 e do teste foi 0,083103. Por alguma razão, as fitness encontradas nos experimentos com 100 gerações ficaram entre 0,06 e 0,09.
- 500 chegou a uma média de 835 segundos por geração e uma fitness média de 0,18568, desvio padrão foi 0,05726, a melhor fitness no treino foi 0,186164 e do teste foi 0,13065.

Apesar do tamanho da população permanecer o mesmo com o teste do número de gerações, quanto mais gerações passam, o tempo por geração também tendeu a aumentar. Isso ocorre provavelmente devido ao consumo de memória.

Levando novamente em questão o tempo/consumo de memória por opção, além de observar com quantas gerações a melhor solução se estabilizou, se tornando a mesma pelas seguintes, conclui-se que o melhor número de gerações para o programa é, também, 50.

Agora, parte-se para o teste das probabilidades. Os testes acima foram feitos com probabilidade de *crossover* 0,9 e de mutação 0,05, e agora testa-se se abaixar de cruzamento para 0,6 e passar de mutação para 0,3 melhora os resultados. Com a população de tamanho 50 e 50 gerações, a média de tempo por geração baixou para 53s por geração, o desvio padrão foi 0,07774, a fitness média foi 0,12376, e a máxima de treino e testes, respectivamente, 0,18616 e 0,17871. Os resultados se mostraram melhores que os obtidos no teste para 50 gerações (que estavam com as probabilidades anteriores e a mesma população), então provavelmente é uma alternativa melhor.

O próximo foi quanto ao tamanho do torneio. Não tiveram grandes alterações nos parâmetros citados nos experimentos anteriores, mas houve um pequeno aumento no tempo por geração e um aumento bom na média da fitness das gerações, então embora não tenha havido aumento considerável na melhor solução, na população em geral, houve.

Assim, recomenda-se aumentar o tamanho do torneio para 5. A média de tempo foi 68 segundos por geração, desvio padrão de 0,05871, fitness de teste média de 0,16944, máxima de treino 0,19738 e de teste 0,1639.

O último experimento foi quanto ao efeito do elitismo. Os anteriores tiveram presença do elitismo, então fez-se experimentos desligando-o. Houve maior variação entre gerações, mas o reultado geral foi, surpreendentemente, melhor. Uma possível explicação é que a falta do elitismo permite uma maior exploração do espaço de soluções, ao invés de focar em um um máximo local. A fitness de teste máxima encontrada foi 0,26499, de treino 0,23163, o desvio padrão foi 0,05289, e a média foi 0,17582. O tempo por geração ficou próximo do experimento anterior.

Os parâmetros escolhidos foram, então, população de 50 indivíduos, 50 gerações, probabilidade de crossover 0,6 e de mutação 0,3, tamanho do torneio 5 e elitismo desligado.

4. Conclusão

O trabalho foi uma boa forma de colocar em prática os conceitos vistos em sala, porém foi difícil a abstração já que faltou abordagem em código durante as aulas. Ainda assim, foi uma boa forma de ver os conceitos concretizados e entender melhor como funcionam algoritmos de programação genética.

Outra dificuldade do trabalho foi a avaliação experimental: mesmo após adaptar para clusters hierárquicos e tentar melhorar o desempenho com operações da numpy e o coletor de lixo de memória, continuou exaustivamente devagar, principalmente para números grandes (como 500 indivíduos ou gerações). Isso tornou o trabalho mais estressante e trabalhoso do que deveria ter sido caso houvesse uma forma melhor de avaliação das soluções (parte do código que causou o tempo excessivo) e tirou grande parte do tempo que poderia ter sido utilizado para melhorar o código a fim de ter eficiência maior e encontrar funções melhores. Tirando esses problemas, foi um projeto muito interessante, possibilitando ver a programação genética em prática, e contribuiu muito para um melhor entendimento prático dos conceitos.

5. Bibliografia

- https://stackabuse.com/hierarchical-clustering-with-python-and-scikit-learn/