UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO CIÊNCIA DA COMPUTAÇÃO

BRENO DE CASTRO PIMENTA
RA: 2017114809

Trabalho: Trabalho Prático 1 – Programação Genética
Disciplina: Computação Natural

Belo Horizonte
2021

1. Introdução

Uma das técnicas amplamente utilizada para a exploração de dados é a chamada *Clustering*, onde segundo Baranauskas (2011) é uma técnica de aprendizado não-supervisionado, utilizada para encontrar padrões e agrupar dados através destas similaridades. Neste trabalho será utilizado uma ramificação dessas técnicas de agrupamento denominada k-means, onde há a verificação da distância dos dados próximos para calcular as similaridades e poder realizar o agrupamento.

O objetivo principal dentro deste trabalho é otimizar a função que calcula as distâncias dos dados dentro do k-means, para isso será utilizado Programação Genética, onde está função será modelada como indivíduo e então ocorrerá um processo de “evolução” do mesmo. Essa evolução se dará da seguinte forma, será criado uma população de indivíduos (candiados a solução) e será aplicado a eles operações genéticas modificando as suas respectivas funções. Após as operações há a verificação da qualidade da solução de cada indivíduo e então há um processo de escolha dos candidatos para a próxima geração e assim sucessivamente até encontrar o candidato que melhor atende ao problema apresentado de cálculo de distância dos dados. No entanto, não basta apenas modelar e evoluir os indivíduos (explicado no item 2), há também que otimizar os parâmetros desta evolução (explicado no item 3), como tamanho da população e proporção das operações.

Essas otimizações e técnicas serão aplicadas a dois datasets, o primeiro representa mulheres com ou sem câncer de mama e o outro representa seis tipos diferentes de vidros, cada dataset fornece vários parâmetros diferentes para que o algortimo possa agrupar os dados.

2. Modelagem

2.1. Indivíduos

O indivíduo será uma representação da função de distância que buscamos otimizar. A estrutura de dados que será usada para sua modelagem é uma árvore estritamente binária (cada nó ou tem dois filhos ou não possui filhos) e os nós podem ser divididos entre terminais e funções. As funções que são descritas logo abaixo, são posicionadas no nós internos da árvore. Já os terminais são divididos em constantes e variáveis e são posicionados na árvore como folhas.

Funções:

- \+ (adição)
- \- (subtração)
- * (multiplicação)
- / (divisão)
- ^ (exponenciação)
- sqrt (raiz)
- log (logaritimo neperiano)
- sin (seno)
- cos (cosseno)

Terminais:

- **Constantes:** São valores reais c arrendodados com duas casas decimais, sendo que -10 < c < 10.
- **Variáveis:** São dados que pertencem ao pontos que buscamos calcular as distâncias.

A classe **SubjectTree** representa o indivíduo descrito acima. Ela é composta por objetos da classe **Node**, que como descrito podem ser divididos entre funções e terminais. É possível imprimir uma representação da função completa de distância que o indivíduo representa utilizando o método printFunction() da classe SubjectTree, um exemplo da saída deste método ao ser chamado com as variáveis {x, y, z}, para um indivíduo de tamanho dois é “(y + x) / (0.2 * z)”. A figura 01 demonstra como a estrutura de dados é construída dentro do objeto para que possa representar este indivíduo.

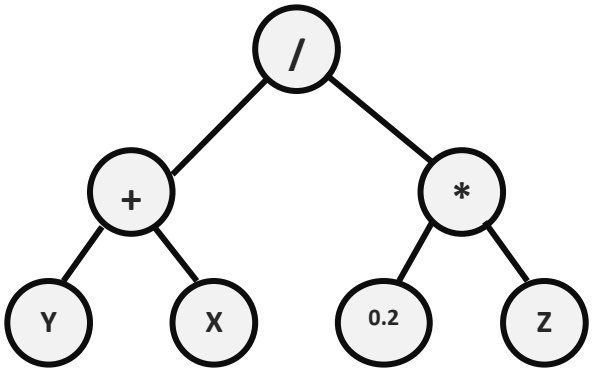

Figura 01 – Estrutura de dados do indivíduo “(y + x) / (0.2 * z)”.

2.2. Inicializando a População

Para a inicialização da população optou-se pela utilização do **Ramped half-and-half** que foi implementado como função dentro do arquivo **inicialization.py**. A implementação se deu da seguinte forma: Sendo T o tamanho da população e M o tamanho máximo dos níveis das árvores que representam os indivíduos, haverá então n indivíduos com cada um dos seguintes tamanhos de árvore {2,3,4...M-1), sendo que haverá também n + (T mod M) indivíduos que serão representados por árvores de tamanho M. A figura abaixo demonstra um exemplo de uma criação de uma população de indivíduos inicializados com T = 4 e M = 3 e com as variáveis x, y, e z.

```
( X sin Y ) * ( Z * X )

( -3.95 sin X ) * ( Y + Z )

( X * Y ) + ( Z * X ) * ( Y - Z ) - ( X + Y )

( X - Y ) sin ( Z cos X ) / ( -0.86 * Y ) + ( Z - X )
```

Figura 02 – Representação de uma população de indivíduos.

Outro fator de grande importância na implementação da inicialização da população foi o uso de **sementes de randomização**. Essa semente é utilizada para a escolha da ordem das variáveis utilizadas e para a escolha da função que terá cada nó interno da árvore que representa o indivíduo. Existe apenas uma semente inicial que é utilizada para criar a população, o que permite que ao executar repetidamente o algoritmo utilizando os mesmos parâmetros se obtenha os mesmos resultados. No entanto, cada indivíduo recebe a semente e uma extensão chamada index, que ao somar as duas cada indivíduo terá uma semente única, criando assim uma variedade de representações diferentes ao inicializar a população.

A randomização é utilizada, como explicado acima, para nós internos, já para as folhas que representam os terminais é um parâmetro de entrada que específica a **probabilidade daquela folha ser variável ou constante**. Caso o parâmetro não seja especificado a relação de probabilidade padrão entre variável e constante é **0.9 e 0.1** respectivamente.

Há também uma relação de **probabilidade para a escolha da função** dos nós internos, possui um valor padrão, porém pode ser passada como parâmetro para a criação dos indivíduos. Essa probabilidade se faz necessária para diminuir a complexidade do algoritmo, ou seja, funções como exponenciação e divisão possuem uma probabilidade menor padronizadamente do que soma e subtração.

2.3. Fitness

A fitness foi implementada como uma função dentro do arquivo **fitness.py**. A função recebe o indivíduo e o utiliza como métrica para treinar um modelo de k-means utilizando o dataset especificado. Após o treino há uma verificação de performance utilizando o **v-score**, que foi implementado utilizando o sklearn.metrics. Ou seja, ao final da fitness o indivíduo terá uma métrica relacionada a ele.

Outras decisões de implementação da fitness consistem no uso de semente de randomização para a inicialização dos centróides do método k-means, esse formato se mostrou muito mais eficiente para a convergência do algoritmo. Além do uso de outra função chamada interlaceLists(), onde concatena a disposição das variáveis dos valores dos pontos passados para a função, fazendo com que as mesmas features fiquem mais próximas nas funções, o que também gerou uma melhoria na performance do algoritmo.

Cabe ressaltar que existe uma possibilidade de melhoria para futuros trabalhos com a implementação de uma verificação de ***bloating*** associada à *fitness*. Essa verificação pode ser implementada através da inspeção de *introns* em cada indivíduo da população, essa inspenção pode ser feita ao retirar-se sistematicamente partes da árvore de representação desse indivíduo e reavaliár-la junto a fitness, caso algum dos resultados alcançados seja o mesmo da árvore anterior, o *intron* então é contabilizado. Para realizar uma medição do *bloating* basta criar uma métrica a partir desta contabilização dos *introns.* Caso haja a necessidade de diminuir o *bloating* durante o processo do algoritmo, basta aplicar o sistema de verificação de *introns* descrito acima e substituir as árvores represntativas dos indivíduos caso encontre uma nova árvore que gere o mesmo resultado na *fitness*.

2.4. Torneio

O torneio foi implementado como função no arquivo **selection.py**. É uma implementação simples da seleção por torneio utilizando elitismo, onde se recebe os indivíduos da última geração e seus resultados na fitness e seleciona os k vencedores, onde k é um parâmetro que foi otimizado durante as experimentações na sessão 3.4.

2.5. Operações Genéticas

As operações genéticas implementadas foram **Mutação de um ponto** e **Crossover**. A mutação foi implementada com o método **mutate()** dentro da classe do indivíduo, onde é escolhida uma camada aleatória da árvore de representação e então é gerado uma chamada que percorre os nós da árvore tomando decisões também aleatórias entre seguir para o ramo da direita ou esquerda. Ao encontrar um nó na camada determinada o algoritmo gera um novo valor para o nó mantendo-se o tipo, ou seja, caso o nó seja um terminal o novo valor será uma variável ou constante, caso sege uma função, outra função será gerada aleatoriamente.

Já o Crossover foi implementado como uma função no arquivo **genetic_diversity.py**, onde recebe dois indivíduos o doador e o donatário. A função chama um método nativo do indivíduo que retorna uma cópia de um nó aleatório de sua árvore de representação, sendo esse nó uma raiz de uma sub-árvore. Então é escolhido de forma aleatória uma posição que essa subárvore pode ser colocada no donatário sem ultrapassar o limite de tamanho de árvore permitida pelo o algoritmo. A implementação foi feita de forma recursiva, pois ao encontrar o nó que será substituído na árvore do donatário o algoritmo então volta ao pai deste nó e muda o ramo para a raiz da subárvore. No crossover tradicional duas árvores doariam entre si, mas na implementação apenas uma árvore fornece e outra árvore recebe, foi uma escolha de implementação, pois se dois indivíduos passarem pelo processo implementado terá o mesmo resultado que um crossover tradicional. E da forma que foi implementada há uma melhoria na abrangência das mudanças realizadas para populações menores ao gerar-se um indivíduo por vez, possibilitando um cruzamento com um número maior de árvores para gerar a mesma quantidade de indivíduos.

3. Experimentos

Todos os experimentos foram salvos em arquivos json e estão localizados na pasta **statistics**, sendo o nome de cada arquivo relacionado com os parâmetros utilizados.

3.1. Tamanho da população

Com os parâmetros estabelecidos em 30 gerações, torneio com k=5 e relação Crossover para mutação de 0.7 para 0.3, variou-se o tamanho da população para buscar o valor ótimo.

Resultados:

- **População = 30: Convergiu em 0,099 na 11º geração**
- **População = 60: Convergiu em 0,197 na 29º geração**
- **População = 100: Convergiu em 0,166 na 15º geração**
- **População = 500: Convergiu em 0,145 na 1º geração**

Diante da qualidade da performance e do custo computacional o número da população será estabelecido em 60.

3.2. Quantida de gerações

Com os parâmetros estabelecidos em 60 indivíduos como população, torneio com k=5 e relação Crossover para mutação de 0.7 para 0.3, variou-se o número de gerações para buscar o valor ótimo. O limite superior para a experimentação foi estabelecido em 60 gerações por causa do alto custo computacional.

Resultados:

- **Gerações = 10: Convergiu em 0,129 na 5º geração**
- **Gerações = 30: Convergiu em 0,197 na 29º geração**
- **Gerações = 45: Convergiu em 0,254 na 36º geração**
- **Gerações = 60: Convergiu em 0,254 na 38º geração**

Diante da qualidade da performance e do custo computacional o número de gerações será estabelecido em 45.

3.3. Proporção de operadores genéticos

Com os parâmetros estabelecidos em 60 indivíduos como população, torneio com k=5 e número de gerações em 45, variou-se a proporção crossover para mutação com objetivo de alcançar o valor ótimo.

Resultados:

- **Crossover = 0.4 e Mutação 0.6: Convergiu em 0,149 na 8º geração.**
- **Crossover = 0.7 e Mutação 0.3: Convergiu em 0,254 na 36º geração.**
- **Crossover = 0.9 e Mutação 0.1: Convergiu em 0,148 na 6º geração.**

Diante apenas da qualidade da performance e a relação dos operadores genéticos estára 0.7 e 0.3 para Crossover e Mutação respectivamente.

3.4. Tamanho do torneio

Com os parâmetros estabelecidos em 60 indivíduos como população, proporção crossover para mutação em 0.7 e 0.3 respectivamente e número de gerações em 45, variou-se o tamanho do torneio com objetivo de alcançar o valor ótimo.

Resultados:

- **k = 2: Convergiu em 0,181 na 37º geração.**
- **k = 5: Convergiu em 0,254 na 36º geração.**
- **k = 7: Convergiu em 0,148 na 5º geração.**

Diante apenas da qualidade da performance e a relação dos operadores genéticos estára 0.7 e 0.3 para Crossover e Mutação respectivamente.

3.5. Parâmetros otimizados:

A maior pontuação encontrada foi 0,254 utilizando a métrica V apresentada na descrição do projeto, para isso foram usados os seguintes parâmetros para o algoritmo:

Resultados:

- **População = 60;**
- **Gerações = 45;**
- **Probabilidade Crossover = 0.7 e Mutação 0.3;**
- **Tamanho do torneio k = 5.**

4. Executando o algoritmo

Como explicado anteriormente todos os resultados são replicáveis por conta do uso das sementes de randomização, no entanto, é imprescindível que para isso os parâmetros estejam como especificado para cada experimento. A semente usada para os experimentos da sessão anterior é zero.

Em ordem de replicar os experimentos basta verificar os parâmetros no início do arquivo **code/main.py** e seguir as instruções de execução que se encontram no arquivo na raiz do projeto chamado **README.txt.**

Ao executar o algoritmo será impresso a função do melhor indíviduo utilizando como variáveis x, y e z, além do v-score desse indivíduo no teste do dataset escolhido.

Será gerado também um arquivo **training_v-score.json** onde estará registrado todos os resultados do treino por geração e seguindo a sguinte ordem: o melhor valor, a média e o pior. Esse arquivo está em um formato pronto para gerar gráficos, porém os gráficos não foram gerados para esta documentção devido ao tempo tomado pelos experimentos.

5. Conclusão

Ao utilizar a distância euclidiana para realizar os mesmo testes que os realizados na sessão 3 deste trabalho, o v-score máximo alcançado é 0,098, sendo que durante os experimentos utilizando da programação genética obteve-se 0,254, ainda com espaço para melhorias como as citadas na sessão 2. Outro resultado surpreendente foi ao gerar os scores para o dataset *glass types* com 6 clusters, esperava-se uma perda de eficiência devido ao aumento da complexidade do algoritmo de clusterização, no entanto o algoritmo gerou um v-score de 0,458 no teste que foi um valor muito além do esperado (os valores do treino deste teste está no arquivo **statistics/glass/training_glass.json**). Logo é possível afirmar que houve êxito em demonstrar a importância e poder desse tipo técnica de programação.

6. Bibliografia

Aulas e slides disponibilizados na matéria.

BARANAUSKAS, José Augusto. **Clustering**. Departamento de Física e Matemática – FFCLRP-USP. São Paulo. 2011. Disponível em <http://dcm.ffclrp.usp.br/~augusto/teaching/ami/AM-I-Clustering.pdf>. Acessado em 25 de jan. de 2020.