Computação Natural

Trabalho Prático I - 2024/2

Igor Lacerda Faria da Silva

1. Introdução

O Trabalho Prático I de Computação Natural consistiu na implementação de uma Programação Genética (PG) para uso como função de distância do algoritmo de *clustering* do `scikit-learn`. A métrica usada na avaliação foi o V-Measure Score (Métrica V). Foram utilizados duas bases de dados como referência: o Breast Cancer e o Wine Red. O repositório deste trabalho pode ser encontrado neste link.

A documentação está divida da seguinte maneira: esta breve introdução descreve o que foi realizado, a Seção 2 descreve os detalhes de implementação, detalhes da representação, *fitness* e operadores utilizados; a Seção 3 é uma análise experimental dos impactos da mudança de parâmetros e a Seção 4 encerra o texto destacando algumas conclusões.

2. Implementação

Esta seção está dividida em duas grandes subseções. A Seção 2.1 apresenta diversas decisões de projeto e inclui instruções para instalação e execução do programa, enquanto a Seção 2.2, discorre em relação às decisões relacionadas à implementação da Programação Genética propriamente dita.

2.1. Arquitetura

A título de completude, esta seção detalha diversos aspectos de *software* da implementação. Primeiramente, a linguagem usada foi Python, versão 3.12.

2.1.1. Instalação

A forma recomendada de instalar as dependências do programa é usando o gerenciador de pacotes uv. Com ele, basta rodar o comando a seguir para instalar as bibliotecas necessárias:

```
uv sync
```

2.1.2. Execução

Existem diversas maneiras de se executar o programa. A principal delas é via linha de comando. No entanto, devido à quantidade de parâmetros do programa, essa opção pode ser massante. De qualquer modo, o comando base, assumindo que o `uv` está sendo usado, é:

```
uv run python -m src ...
```

Em que as reticências denotam os parâmetros. Ao todo, existem 13 parâmetros, majoritariamente obrigatórios. A seguir, estes são listados:

- `--seed`, semente aleatória
- `--leaf-probability`, probabilidade de se gerar uma folha ao construir a árvore
- `--mutation-probability`, probabilidade de mutação
- `--swap-terminal-probability`, probabilidade de se trocar um terminal na mutação
- `--swap-operator-probability`, probabilidade de se trocar um operador na mutação
- `--crossover-probability`, probabilidade de cruzamento
- `--population-size`, tamanho da população

- `--tournament-size`, tamanho do torneio
- `--elitism-size`, tamanho do elitismo (tamanho 0 significa ausência de elitismo)
- `--max-generations`, quantidade de gerações
- `--max-depth`, profundidade máxima da árvore
- Banco de Dados (não possui *flag*). Deve ser especificado como `wine_red` ou `breast_cancer_coimbra`

O parâmetro restante controla se a saída deve ser verbosa ou não. A saída verbosa inclui as "Estatísticas Importantes" da especificação, enquanto que a versão não verbosa apenas imprime a Métrica V dos dados de teste. A corretude dos parâmetros é vastamente checada, por exemplo: o programa retorna uma exceção caso o tamanho do elitismo seja maior do que o número de indivíduos na população.

Devido à quantidade exorbitante de parâmetros, foram desenvolvidas algumas alternativas para execução. Apesar disso, para que elas funcionem corretamente, é necessário ter seguido os passos da Seção 2.1.1. A alternativa mais simples é rodar o programa pelo VS Code, através da configuração de depuração. Ela atribui aos parâmetros valores inspirados na especificação, que descreve parâmetros iniciais para começar os experimentos[1]. Esta foi a principal forma de execução durante o desenvolvimento.

A segunda alternativa recomendada é através do Jupyter. A partir do módulo `loader`, é definida uma configuração padrão para o programa (uma vez que não é possível passar os parâmetros diretamente via linha de comando no Jupyter). A finalidade dessa forma de execução foi permitir a visualização dos operadores genéticos, mas ela também pode ser usada para rodar o programa por completo.

Além disso, o programa também conta com alguns testes de unidade, com uso do *framework* pytest[2]. Enquanto o Jupyter foi usado para testar funcionalidades de "mais alto nível", os testes foram desenvolvidos para garantir que comportamentos mais simples não fossem "quebrados" no decorrer do projeto. Os testes podem ser executados com o comando

```
uv run python -m pytest
```

Para concluir, na fase de experimentação, foram desenvolvidos alguns *scripts* para facilitar a execução repetida do programa. Os *scripts* foram implementados para o *shell* fish e não são compatíveis com o bash e afins. Eles se encontram no diretório `./run`, na raiz do projeto (mais detalhes na Seção 2.1.3).

2.1.3. Estruturação dos Arquivos

O programa está dividido em módulos. A ideia era simplificar o *loop* principal do algoritmo, abstraindo detalhes de implementação. Assim, o *loop* principal chama uma função `select()` (que faz a seleção), implementada em outro arquivo (por exemplo). Essa modularidade torna o programa mais extensível, pois seria relativamente simples trocar as funções que implementam o cerne do algoritmo. Devido à modularização, ao todo são 18 arquivos de código-fonte[3].

---

[1] Por exemplo, probabilidade de cruzamento igual a 90%. Alguns parâmetros foram escolhidos arbitrariamente, por falta de referência. Para detalhes, consultar o arquivo.

[2] Que é uma das dependências do projeto, então não é necessário instalá-lo separadamente.

[3] Excluindo arquivos de teste e afins.

A estruturação do repositório como um todo (para além do código-fonte) também traz essa modularidade. Além do código fonte que se encontra na pasta `./src` (e desta documentação, em `./docs`), o projeto também conta com:

- `./test`, com os arquivos de teste;
- `./scripts`, com alguns programas simples para auxiliar na agregação de estatísticas e geração de gráficos;
- `./run`, com alguns *scripts* para execução do programa (ver final da Seção 2.1.2);
- `./data`, contém os *datasets Breast Cancer* e *Wine Red*;
- `./dumps`, com os relatórios brutos das execuções usadas para a Seção 3;
- `./assets`, contém as figuras e estatísticas agregadas usadas na Seção 3.

Ademais, alguns *notebooks* do Jupyter estão espalhados na raiz.

2.2. Programação Genética

Essa seção descreve as escolhas de implementação associadas à Programação Genética.

2.2.1. Preliminares

Antes de iniciar a discussão sobre a PG, duas decisões cruciais foram tomadas em relação a:

- Normalização dos Dados. Para evitar efeitos provenientes da diferença de escala entre as colunas, os dados foram normalizados. Mais especificamente, foi realizada a norma `Max`, por coluna. Assim, os valores de cada coluna variam entre 0 e 1, sendo que sempre vai existir ao menos um valor 1 por coluna. Outros métodos estatísticos foram considerados, como a transformação de Box-Cox, mas eventualmente foram descartados (por exemplo, o Box-Cox introduz valores negativos que poderiam afetar o desempenho da PG).
- Cálculo das Distâncias. Para simplificar o cálculo da distância, a função gerada pela PG é, primeiro executada para uma instância do *dataset* e depois para outra. A distância final entre as instâncias é o valor absoluto da diferença. A base para essa decisão foi exclusivamente a simplicidade de implementação.

2.2.2. Representação de Indivíduos

Optou-se por representar um indivíduo como uma árvore binária, com suporte às quatro operações básicas (adição, subtração, divisão[4] e multiplicação) e com os terminais sendo as *features* do *dataset*. Tal como a escolha do cálculo das distâncias, essa opção foi guiada pela simplicidade de implementação (apesar de pecar em poder de representação). Somente com os operadores básicos (operadores binários, no geral), os operadores genéticos são vastamente simplificados (mais detalhes na Seção 2.2.5).

A implementação da árvore não dependeu de bibliotecas. A árvore é gerada aleatoriamente (ver Seção 2.2.3) e, para se realizar a avaliação de uma dada coluna do *dataset*, é realizado um caminhamento na árvore, substituindo os terminais pelos valores da coluna. A árvore também conta com algumas operações de auxílio, como as funções `height`, `depth` e `traverse`. Além disso, possui um conversor para representação gráfica usando o Graphviz[5].

[4] Naturalmente que foi usada a divisão protegida, substituindo 0 por um ε.

[5] Na seção de instalação não foi comentado, mas para usar essa representação, é necessário instalar o Graphviz separadamente.

2.2.3. Geração de Indivíduos

A geração de indivíduos também foi a mais simples possível. A abordagem utilizada é similar ao método *grow*, visto em sala de aula. Cada nó tem uma chance de ser um terminal (ou é garantido que será um terminal se for a profundidade máxima), caso contrário é escolhido um operador e seus filhos são gerados recursivamente. Uma abordagem mais balanceada, como o *ramped half-and-half* não foi considerada, por introduzir uma certa complexidade. Similarmente, o método *grow* introduz complexidade pois adiciona mais um parâmetro a mais no algoritmo: a chance de uma folha ser gerada (`--leaf-probability`, da Seção 2.1.2).

2.2.4. Seleção

Como definido na especificação, o método de seleção foi o torneio. A única questão notável aqui foi o esquema de substituição: os pais sempre substituem os filhos. Ou seja, não foi usado *crowding* ou mecanismo similar para lidar com diversidade.

2.2.5. Operadores Genéticos

2.2.5.1. Cruzamento

O cruzamento implementado seleciona um nó de cada árvore e faz a troca. Como a árvore é bem simples, não é necessário preocupação com a propriedade de fechamento. O único invariante que deve ser respeitado é a profundidade da árvore, que deve ser menor que o tamanho máximo do indivíduo. Para checar isso, as funções auxiliares mencionadas em Seção 2.2.2 são usadas. Mais especificamente, confere-se se a soma da altura de nó com a profundidade do outro excede o limite. Como a árvore binária possui um nó para indicar o pai, certo cuidado é tomado para realizar corretamente a troca.

As figuras a seguir, geradas com base na representação do Graphviz (ver Seção 2.2.2), ilustram um exemplo de cruzamento. Nelas, os nós destacados de vermelho indicam a troca.

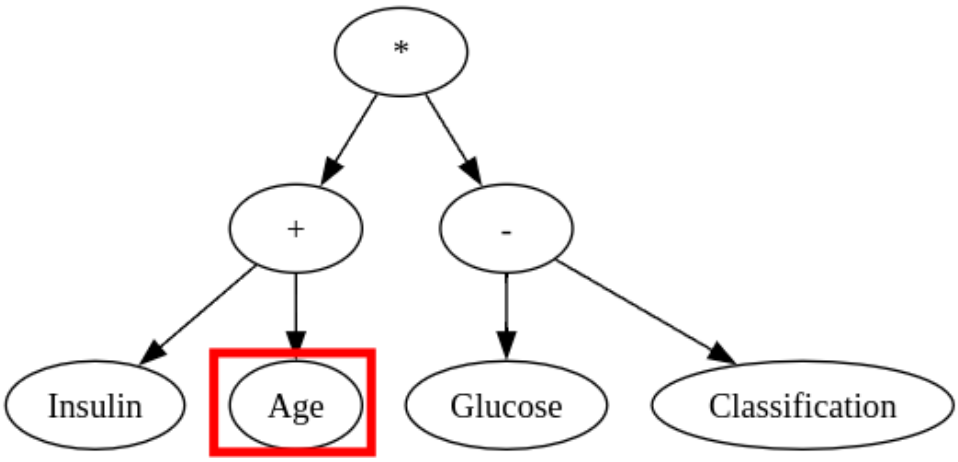

Figura 1: Pai 1

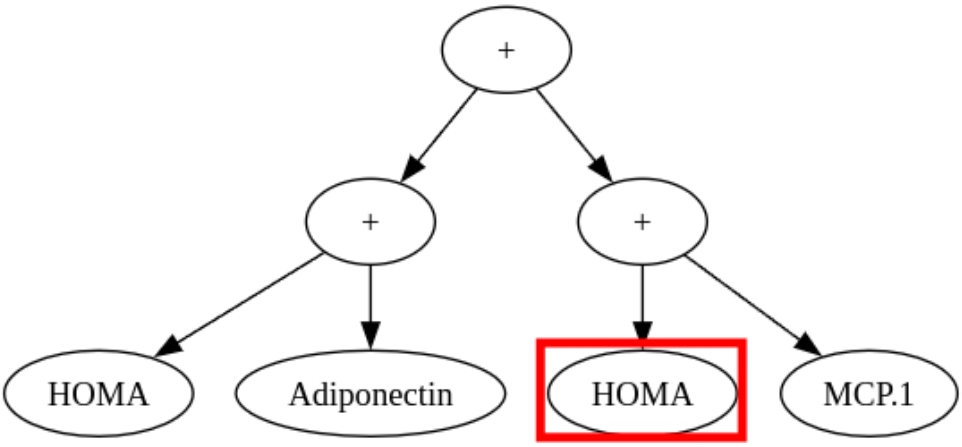

Figura 2: Pai 2

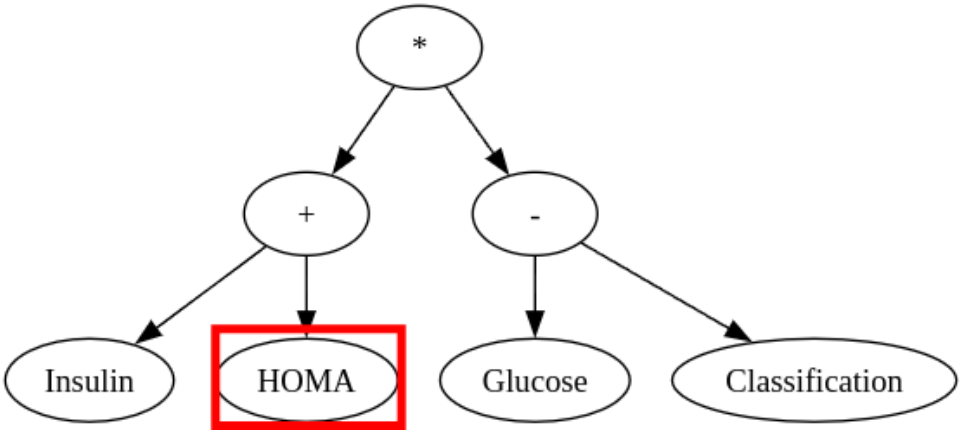

Figura 3: Filho 1

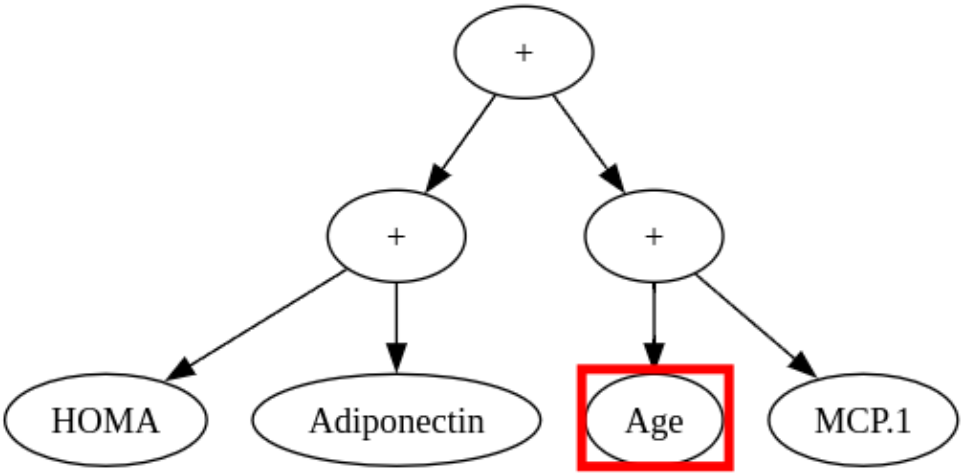

Figura 4: Filho 2

2.2.5.2. Mutação

2.2.6. Fitness

Além do comentário sobre o cálculo das distâncias da Seção 2.2.1, não há muito o que comentar sobre sua implementação. Tentou-se realizar o cálculo das distâncias com programação

paralela (via Numba), mas não foram obtidos bons resultados[6]. De qualquer modo, o Numba foi usado para compilar a função de cálculo de distâncias, o que tornou a função mais rápida (a diferença é mais notável para instâncias maiores).

O cálculo da *fitness* funciona da seguinte maneira: primeiro, as avaliações das colunas são geradas, depois é calculada a matriz de distâncias (com base nas avaliações) e é feito um `fit_predict` no `AgglomerativeClustering` do `scikit-learn`. Por fim, é realizado o cálculo da Métrica V, comparando-se com as classes reais.

3. Experimentos

O roteiro de experimentação foi baseado

4. Conclusões

5. Bibliografia

[6]Outra possibilidade de melhoramento de performance que não se mostrou promissora, foi o uso do compilador pypy.