Primeiro Trabalho Prático de Algoritmos II

Marcelo Mascarenhas Ribeiro de Araújo Araújo

Janeiro 2022

1 Introdução

Essa documentação tem a finalidade de apresentar diretrizes a serem seguidas na execução do programa, além de fornecer uma breve explicação da implementação realizada, em concomitância com uma análise de complexidade das principais funções utilizadas para a solução do problema proposto.

O projeto trata da implementação da técnica de K-Nearest-Neighbours (KNN), através do uso de KDTrees. Nesse sentido, o projeto modela o problema computacional descrito e apresenta uma solução viável, que recebe os parâmetros através de um arquivo .dat, e realiza a leitura e a execução seguindo a lógica proposta na apresentação dos requerimentos do projeto.

Para informações mais detalhadas sobre o projeto, consulte as seções a seguir, sendo que:

- A seção 2 trata sobre as decisões tomadas para a resolução do problema proposto, bem como alguns detalhes de implementação, além do ambiente computacional que o trabalho foi desenvolvido.
- A seção 3 informa as instruções de compilação e execução.
- A seção 4 exibe algumas avaliações experimentais que foram utilizadas em alguns datasets da base de dados KEEL para mensurar a eficiência do dataset.
- A seção 5 conclui a documentação, fornecendo alguns detalhes adicionais sobre o processo de desenvolvimento e considerações em relação à solução proposta.

2 Implementação

2.1 Especificações

O programa foi desenvolvido e testado nas seguintes especificações:

- Linguagem: Python.
- OS e Versão do Sistema: POP_OS!, versão 21.10.
- Configuração do computador: AMD Ryzen 5 3600, 16GB de RAM.

2.2 Descrição da Solução do Problema

Para solucionar o problema proposto, primeiramente foi necessário a implementação de uma KDTree. Uma KDTree é uma estrutura de dados especial, cuja ideia principal é particionar um espaço de qualquer dimensão, de modo que cada ponto se localize em uma das partições[1]. O particionamento é realizado através de cortes intercalados dimensionalmente de maneira cíclica. Cada corte é determinado através do valor da mediana dos pontos que em uma dada dimensão, considerando somente o conjunto de pontos que se localizam na região do espaço que está sendo particionada. Dessa maneira, essa estrutura é representada computacionalmente como uma árvore, onde cada nó representa um corte em um determinado eixo, e cada folha representa um ponto no espaço.

Construída a árvore, podemos realizar buscas de um ponto mais próximo de um ponto alvo específico, *p1*. A primeira etapa é achar a partição que o *p1* se localizaria, e medir a distância euclidiana do *p1* até o ponto que está na partição encontrada, *p2*. Se existir uma divisória da partição no alcance da circunferência formada com o centro sendo *p1* e o raio a distância de *p1* e *p2*, então é necessário checar a partição subjacente que compartilha a divisória com a primeira partição, já que pode existir um ponto mais próximo de *p1* em uma outra partição. Caso contrário, *p2* é o ponto mais próximo. Podemos generalizar o raciocínio com K pontos, com a diferença que o algoritmo checará as partições de maneira recursiva até achar os K pontos, e o raio de busca será determinado pela distância entre *p1* e o ponto mais longínquo dentre os pontos candidatos.

2.3 Fluxograma

Nesse sentido, o fluxograma do programa segue a seguinte lógica: primeiro, os dados do dataset são lidos e algumas informações armazenadas. Após esse processo, o dataset é dividido em um conjunto de treino e teste, respeitando a proporção de 70-30, determinada no requerimento do trabalho. Depois disso, é criada uma instância da classe KNN, que recebe um conjunto de pontos de

[1] No caso dessa implementação específica, onde o leafsize=1. Existem implementações onde o número de ponto nos nós folhas é maior que 1. Isso diminui a profundidade da árvore, mas aumenta o custo total de comparação no momento da busca por pontos.

treinamento, e monta a KDTree. Após isso, é chamada a função classify, que busca os K pontos mais próximos de cada um dos pontos de teste. Após a classificação, o programa tira métricas de acurácia, precisão e revocação, e encerra.

2.4 Estrutura de Dados e Classes

O programa contém três classes principais. Aqui será fornecido uma breve explicação. Para mais detalhes, consulte o código que acompanha a documentação.

A primeira classe, denominada "Keel", é responsável por processar os dados do arquivo *.dat*. A classe manipula os arquivos do formato das bases de dados oriundas do repositório KEEL. Além disso, é responsável por fazer o split dos dados em treino e teste, na proporção requisitada na especificação do trabalho. Como os dados são armazenados numa matriz, gerada através do uso da biblioteca *numpy*, a separação foi feita embaralhando as linhas da matriz, calculando o piso de 70% do número de linhas, e realizando a quebra.

A segunda classe é denominada de "Kdtree", e contém os trechos de código responsável por construir a KDTree, a partir de um conjunto de pontos. Primeiro é realizado um pré-processamento nos dados, aglutinando pontos de mesmas coordenadas e classes. Para registrar o número de pontos que foram fundidos, é adicionada uma coluna de pesos. Pontos que não possuiam duplicatas recebem peso 1 (um), enquanto pontos que possuiam duplicatas recebem valores equivalente à quantidade de pontos iguais. É importante salientar que apenas pontos com exatamente o mesmo rótulo são juntados. Pontos de mesma coordenada, mas com classes diferentes, permanecem separados. Essa coluna de pesos é considerada apenas na avaliação dos rótulos dos KNN de um ponto de teste, não sendo relevante para a construção da árvore.

Após essa etapa, a árvore é construída. Uma classe interna, denominada de "Node", serve como alicerce para a árvore, que armazena a coordenada de corte(mediana) de uma determinada dimensão, um nó esquerdo e um nó direito. A cada iteração, a matriz de entrada é ordenada com respeito à uma dimensão específica, e partida ao meio. Isso é valido pois, um conjuntos de pontos ordenados têm, por definição, o valor da mediana na metade do vetor. Esse processo é repetido até que as matrizes tenham eventualmente um elemento. Nesse caso, o ponto é atribuído ao nó esquerdo ou direito de um "Node", se tornando folhas.

A terceira classe é denominada de "Knn", e estabelece uma relação de herança com a classe Kdtree. Ou seja, objetos da classe KNN possuem Kdtrees. Porém, a classe KNN possui o método de busca dos KNN e da produção das métricas de avaliação. Para buscar os KNN, é realizada uma busca recursiva, e é utilizada uma lista de tuplas para armazenar os pontos candidatos, onde eles são ordenados de maneira crescente em relação à distância. Tendo em vista que o python utiliza um algoritmo extremamente eficiente de ordenação, denominado

Timsort[2], a complexidade de ordenação dos K elementos dado a inserção de um novo ponto candidato não aumentou significativamente o tempo de execução, ao menos empiricamente.

As métricas de acurácia, precisão e revocação foram obtidas através do cálculo da matriz de confusão, obtida da comparação com os resultados preditos e reais do conjunto de teste.

3 Instruções de execução

Primeiro é preciso realizar o download do repositório, através do seguinte comando:

git clone https://github.com/marcelo-mascarenhas/KNN

Depois dessa etapa, para executá-lo, é suficiente executar a seguinte linha de comando:

python3 main.py -inf caminho_para_dataset –k numero_de_vizinhos

O caminho do dataset deve ser completo, e ele deve estar no mesmo formado dos datasets do repositório KEEL[3]. Para mais informações, execute:

python3 main.py -h

4 Resultados com Datasets KEEL

Foram realizados 10 testes com as bases de dados oriundas do repositório KEEL. Foram medidos apenas datasets com duas dimensões, pois a função responsável por calcular as métricas de precisão, acurácia e revocação não foi adaptada para dataset multiclasses. Os valores foram arredondados.

Dataset	Acurácia	Precisão	Revocação	K
Bananas	0.91	0.85	0.83	10
Haberman	0.92	0.73	0.76	5
Ionosphere	0.6	0.82	0.92	3
Twonorm	0.97	0.96	0.96	5
Phoneme	0.9	0.85	0.90	10
Magic	0.93	0.79	0.78	7
Coil2000	0.98	0.93	0.93	10
Australian	0.77	0.64	0.66	10
Heart	0.69	0.62	0.68	10
Appendicitis	0.91	0.81	0.85	10

[2]https://ie.nitk.ac.in/blog/2019/11/29/timsort-the-fastest-sorting-algorithm/
[3]https://sci2s.ugr.es/keel/category.php?cat=clas

5 Conclusão

Considerando todas as informações supracitas, o problema proposto foi resolvido de maneira satisfatória. Não foram observados erros de execução nos testes realizados. Além disso, a resolução se mostrou extremamente modularizada e não tão custosa, apesar de provavelmente existirem projetos com uma eficiência maior.