Trabalho Prático I – Algoritmos II

Vitor Rodarte Ricoy - 2019007112

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
Repositório no Github: https://github.com/vitorricoy/TP1_ALG2

1. Introdução

O trabalho proposto tem como objetivo praticar e fixar os conceitos aprendidos em aula relativos à implementação de uma Árvore K-Dimensional (Árvore KD), além de conceitos de geometria computacional no geral. Esse trabalho consiste na implementação de um classificador baseado nos K vizinhos mais próximos, utilizando a estrutura da Árvore KD, que permite uma implementação eficiente desse algoritmo, que é geralmente usado para classificação.

2. Descrição Geral

A implementação do trabalho foi feita em Python 3.9.6, utilizando apenas bibliotecas padrões do Python. Foram definidas três classes: uma classe *No*, que representa um nó da Árvore KD; uma classe *ArvoreKD*, que representa a estrutura de dados da Árvore K-Dimensional; e uma classe *xNN*, que representa o classificador baseado nos K vizinhos mais próximos.

2.1. Árvore KD

A Árvore K-Dimensional foi implementada de acordo com o que foi visto nas aulas. Primeiramente foi definida a estrutura para os nós da árvore, por meio de uma classe. Cada nó contém um valor, uma profundidade e uma identificação se o nó é uma folha. O nó também pode possuir um filho à esquerda e um filho à direita. Vale notar que o valor do nó possui uma estrutura diferente dependendo do nó ser folha ou não. Caso ele seja uma folha, o valor é um ponto, já caso ele seja um nó intermediário o valor é a mediana da dimensão analisada, usada para dividir os nós em dois grupos.

A Árvore KD é construída pelo mesmo processo visto em aula. Primeiramente é escolhida a mediana dos valores da primeira coordenada dos pontos analisados e eles são divididos em dois grupos: um que tem esse valor menor ou igual o da mediana e outro que tem esse valor maior do que a mediana. Esse processo é repetido para todos os grupos e todas as coordenadas do ponto, até que se tenha grupos que contêm apenas um ponto. Quando isso acontece esse ponto é inserido na árvore como uma folha. Esse processo garante que a árvore terá um certo balanceamento já que cada conjunto de pontos dividido possui o mesmo tamanho ou apenas um ponto a mais ou a menos, caso se tenha um número ímpar de pontos.

Após termos a árvore construída, podemos consultar a sua estrutura para realizar operações com os pontos considerados em sua construção. Mais especificamente desejamos encontrar os k vizinhos mais próximos de um ponto que pode estar ou não contido na estrutura. A utilização da Árvore KD para esse propósito foi implementada no classificador KNN e será explicada a seguir.

2.2. Classificador KNN

O classificador KNN foi feito seguindo o que foi descrito na especificação do trabalho. Quando o classificador é inicializado, é executada uma função de classificação para cada ponto que deve ser classificado.

A função de classificação de um ponto se inicia da raíz da Árvore KD. Primeiramente percorremos a árvore como se tivéssemos inserindo o ponto a ser classificado. Após chegar a uma folha na árvore, sabemos que essa folha representa um ponto próximo ao ponto classificado e salvamos a distância entre eles em uma fila de prioridades. Sempre que um nó folha é visitado a fila de prioridades é atualizada com a nova distância encontrada, sendo que caso a fila contenha mais do que k elementos, o ponto de maior custo é descartado para que ela não exceda esse tamanho.

Após encontrar um nó folha, a recursão feita na árvore começa a retornar, realizando uma "subida" na árvore. A cada "subida", é verificado se o conjunto que foi descartado naquele passo da recursão pode conter pontos mais próximos do que a distância máxima encontrada até o momento, ou seja, caso na recursão inicial foi visitado o filho à esquerda, após retornar dela é verificado se os pontos definidos pelo filho à direita podem estar a uma distância menor que a distância máxima encontrada. A lógica análoga acontece para a visitação do filho à direita.

Para verificar se os pontos de um filho podem estar a uma distância menor que a distância máxima foi implementada a lógica descrita no enunciado do trabalho: é traçado uma hiperesfera a partir do ponto sendo classificado, com raio igual à distância máxima, e é verificado se essa hiperesfera possui alguma interseção com o hiperplano definido pelo filho sendo analisado.

Vale notar que cada filho define um hiperplano de acordo com o valor da mediana escolhida para dividir a coordenada naquela profundidade da Árvore KD, portanto existem dois cenários para verificar a interseção: o hiperplano está à direita do ponto ou o hiperplano está à esquerda do ponto. No primeiro caso, basta verificar se a coordenada do ponto sendo classificado somada do raio é maior ou igual ao valor da coordenada que define o hiperplano. Já no segundo caso, analisamos se o valor da coordenada do ponto subtraído do raio é menor ou igual ao valor da coordenada que define o hiperplano.

Caso seja constatado que o conjunto descartado inicialmente pela recursão pode conter pontos mais próximos, a função é executada normalmente para esse conjunto de pontos, ou seja, ambos os filhos desse nó da árvore serão visitados. A visitação desse filho que, anteriormente foi descartado, segue com o mesmo funcionamento da visitação do filho que já foi visitado, ou seja, é seguido o comportamento de inserção na árvore do ponto sendo classificado até alcançar uma folha, onde é atualizada a fila de prioridades que contém as k menores distâncias encontradas.

Vale notar que a função recursiva descrita foi implementada com algumas otimizações para economizar tempo de execução. Uma delas foi a criação de diversos parâmetros da função como atributos da classe, para evitar desses parâmetros serem passados a cada chamada recursiva. Outra otimização foi salvar na fila de prioridades um id para cada ponto, ao invés do ponto em si, para evitar manipulação das tuplas que representam os pontos a cada atualização da fila.

Depois da recursão descrita anteriormente ser finalizada, temos na fila de prioridades a lista dos k vizinhos mais próximos do ponto. Com essa lista, podemos classificar o ponto com

um rótulo A ou B, de acordo com o rótulo mais frequente nos pontos da lista retornada. Vale notar que foi decidido que caso tenha um empate entre a frequência dos rótulos A e B, o rótulo do ponto mais próximo é escolhido.

Após a classificação de todos os pontos, são salvas as informações necessárias para o cálculo das estatísticas, e em seguida, as estatísticas em si são calculadas e ficam disponíveis para serem lidas da classe.

2.3. Programa Principal

O programa principal é responsável por ler os arquivos de testes fornecidos, separar o conjunto de treinamento e de teste de forma aleatória na proporção de 30-70, executar o classificador e imprimir os resultados obtidos por ele. O programa principal implementado segue as decisões explicadas na seção de testes.

Vale notar que como os arquivos de testes podem conter atributos não numéricos foi adotada uma representação de atributos nominais em que eles são substituídos por um inteiro. Ou seja, quando um atributo nominal é encontrado ele recebe um valor, e toda vez que ele aparece novamente esse mesmo valor é atribuído a ele. Essa representação foi escolhida por ser mais simples, mesmo que possa gerar resultados ruins devido a essa conversão arbitrária de atributos nominais.

3. Testes

Foram escolhidos os seguintes conjuntos de testes, retirados do site sugerido na especificação do trabalho: *adult*, *banana*, *chess*, *hepatitis*, *magic*, *mushroom*, *pima*, *ring*, *titanic* e *twonorm*.

Os valores apresentados são as médias de 10 execuções para cada conjunto de testes, já que o conjunto de treino e teste são aleatorizados.

Os testes foram executados com a implementação PyPy do Python para reduzir o overhead causado pela linguagem, permitindo tempos de execução menores. As execuções do meu algoritmo foram comparadas com os resultados do classificador da classe *KDTree* implementada na biblioteca *sklearn*, com exceção do tempo devido ao grande número de otimizações feita na estrutura da biblioteca, o que reduz a relevância da comparação de tempo.

Os arquivos de testes foram modificados para não conter os metadados representados por tags no seu início. Isso foi feito para facilitar a leitura desses arquivos pelo programa. Também foi inserido o valor de k escolhido para a instância de teste no início do seu arquivo. Para todos os programas, o valor de k escolhido foi o resultado da execução de um script que usa a biblioteca *sklearn* para determinar um valor de k para o conjunto de dados.

Por fim, os arquivos de código usados para realizar os testes foram entregues em uma pasta *codigo_auxiliar_testes*, enquanto os arquivos do conjunto de testes foram entregues em uma pasta *testes*. Seguem as execuções dos testes:

			Estatísticas da Implementação				Estatísticas de Controle		
Conjunto de Teste	**k**	**n**	**Tempo de execução (s)**	**Acurácia**	**Revocação**	**Precisão**	**Acurácia**	**Revocação**	**Precisão**
adult	16	48842	221,066	0,7442	0,9767	0,8022	0,7364	0,9697	0,8024
banana	70	5300	0,053	0,3413	0,6183	0,5911	0,3845	0,6975	0,6168
chess	4	3196	0,588	0,4983	0,9522	0,9350	0,4756	0,9069	0,9710
hepatitis	11	155	0,014	0,7913	0,9921	0,8004	0,7717	0,9973	0,7735
magic	3	19020	1,434	0,6449	0,9969	0,9945	0,6474	0,9971	0,9946
mushroom	3	8124	1,672	0,4849	0,9999	1,0000	0,4847	1,0000	0,9997
pima	49	768	0,0635	0,0639	0,1861	0,6330	0,0513	0,1441	0,6847
ring	84	7400	3,468	0,4569	0,9235	0.5166	0,4556	0,9214	0,5156
titanic	5	2201	0,059	0,9715	1,0000	0,9857	0,9729	0,9985	0,9871
twonorm	98	7400	1,878	0,2594	0,5192	0,5012	0,2450	0,4927	0,5051

Primeiramente podemos notar que os valores de acurácia, revocação e precisão da minha implementação e do algoritmo de controle estão próximos, o que é uma indicação de que o algoritmo implementado funciona. As diferenças podem ser explicadas pela divisão aleatória de conjunto de treino e de teste e também pelo comportamento quando se tem um "empate" na escolha do rótulo de um ponto.

Um fato que podemos notar é o tempo de execução. O tempo se comporta bem com relação ao crescimento de n e de k, sendo que ele sofre um aumento considerável em conjunto de testes em que precisamos visitar os dois filhos de um vértice da árvore com mais frequência, que é o caso do teste contido no arquivo *adult*. Isso explica o tempo bem mais longo de execução do arquivo *adult*, enquanto arquivos com muitos vértices, como o magic, possuem um tempo de execução bem menor.

Finalmente, a comparação do comportamento das estatísticas de desempenho do classificador nos leva à conclusão que as duas implementações se comportam de forma similar, já que em alguns momentos a minha implementação obtém resultados marginalmente melhores e em outros momentos a implementação da biblioteca obtém resultados marginalmente melhores. Essa diferença se explica pelos fatos citados no primeiro parágrafo.