Trabalho Prático 1: Aplicações de algoritmos geométricos

Gabriel Oliveira[1]

[1]Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
2019041523

gabrieloliveira@gmail.com

1. Introdução

Neste trabalho implementamos uma kd-tree que retorna os k prontos mais próximos do conjunto de teste, e um sistema de classifica k-NN que usa a kd-tree dado um ponto usa uma kd-tree para encontrar os k pontos mais próximos e classifica o pontoa traves do voto majoritário e calcula as métricas da classificação.

2. Implementação

O problema foi desenvolvido em C++.

Nosso programa tem uma lista de arquivos para rodar que ele deve rodar o algorítimo, para cada arquivo foi pre processado para facilitar a leitura pelo ifstream, eles sempre começam com o número de dimensões e o número de classes seguido por múltiplas linhas com os pontos e as classes sempre são inteiros começando do 0. Os pontos são colocados em um vetor.

Para conseguir métricas mais precisas embaralhamos o vetor de pontos 10 vezes e tiramos a média das métricas do algorítimo. O embaralhamento faz com que cada vez o 70% dos pontos separados para o conjunto de treino seja diferente. Nós inicializamos o k-NN passando o número de dimensões, o número de classes e o conjunto de treino. O número de dimensões e o conjunto de treino são usados para inicializar a kd-tree e o número de classes é utilizado na hora de calcular as métricas.

Depois passamos um número e o conjunto de teste para o kNN para calcular as métricas, o número é a quantidade de pontos mais próximos a ser utilizado para classificar o conjunto de teste.

2.1. ponto

O struct ponto(chamando de point no código) é um struct com um vetor de doubles que guarda as coordenadas e um inteiro que guarda a classe do ponto.

2.2. k-NN

A classe k-NN(chamada por extenso de kNearestNeigbour no código) tem uma árvore kd e um inteiro que guarda o número de classes.

A função initialize guarda o número de classes e inicializa a árvore kd com o conjunto de treino.

A função classify recebe o número de pontos mais próximos que deve ser usado e o conjunto de teste e retorna os pares dos pontos e a classificação deles. Para isso chamamos

a querry da árvore kd e simplesmente contamos qual é a classe que mais apareceu dos pontos mais próximos. Em caso de empate pegamos aquela que tem o ponto mais distante mais próximo.

A função metrics recebe o número de pontos mais próximos e o conjunto teste e usa a função classify para prever a classificação dos pontos, em seguia produz a matriz de confusão. Depois usamos ela para calcular a acurácia de todos os pontos e a precisão e revocação de cada classe(como mostrado nesse link: https://stats.stackexchange.com/questions/51296/how-do-you-calculate-precision-and-recall-for-multiclass-classification-using-co). Depois retornamos as metricas como um vetor de triplas com as metrica de cada classe em cada posição.

2.3. arvore kd

A classe arvore kd(chamada por extenso de kdtree no codigo) tem um vetor de pontos que guarda os nos externos da árvore e um vetor de doubles que guarda os planos dos nos internos da árvore. Nenhum dos nos possuem ponteiros para outros nos, o código consegue decidir quais são os filhos de quem puramente a partir das posições dos pontos no vetor(isso vai ser explicado na próxima subseção). A arvore kd também tem um inteiro para guardar a dimensão. Para não ter que ficar passando de novo os pontos para toda chamada recursiva da querry, a árvore kd possui um inteiro e uma lista de prioridade como variáveis. Elas são usadas apenas durante a querry e existem para diminuir o número de parâmetros que precisam ser passados na recurção.

A função initialize recebe o número de dimensões dos pontos e o conjunto de treino, guarda o número de dimensões, coloca o conjunto de treino no vetor de nos externos, muda o tamanho do vetor dos nos internos e chama a função build para a raiz na dimensão 0.

A função build não precisa receber a lista de pontos para saber quais pontos são filhos do ponto atual(isso já esta codificado da maneira que passamos o índice do no), mas aqui vou explicar de maneira simplificada como o código funciona, para isso assuma que todos os nos já tem os filhos corretos da maneira desejada(ou seja, o número de folhas do filho da esquerda é o mesmo ou um a mais do que o da direita e tudo o que falta é colocar os pontos na posição correta). O código seria esse:

build (no, $pontos$, d) **:**
  $d = d\%dim$
  if $|pontos| == 1$ **then**
    $no.ponto = pontos[0]$
    return
  else
    $pontosMenores, pontosMaiores, no.plano =$
      $quickselect(pontos, \lfloor |pontos|/2 \rfloor, d)$
    $build(no.esq, pontosMenores, d+1)$
    $build(no.dir, pontosMaiores, d+1)$
    return
  end

Neste código usamos o quickselect, ele nos permite encontrar o ponto mediano(em relação a dimensão passada) em $O(n)$ no tempo medi-o. Ele também separa os pontos menores e maiores em dois conjuntos(na implementação minha a própria "ordenação" dele já faz com que os conjuntos sejam separados). Observe que o nosso algorítimo é uma generalização do algorítimo do livro Computational Geometry. O algorítimo alterna entre dimensões em cada chamada recursiva caso exista apenas um ponto ele cria uma folha, caso contrario ele separa os pontos conforme a dimensão atual e defini o plano e constrói o filho esquerdo e direito.

A função querry recebe o número de pontos a ser selecionados e um ponto, e retorna a lista de pontos do tamanho correto(igual ao parâmetro passado) dos pontos mais próximos do ponto passado, ela simplesmente reseta a fila de prioridade guarda na classe o número de pontos a ser selecionados e o ponto, e chama uma função privada chamada _querry passando o no interno como parâmetro. Esta função coloca os pontos mais próximos na fila de prioridade, depois o querry coloca eles em um vetor e os retorna.

A função _querry usa uma função push que coloca o ponto na fila de prioridade se ela tiver espaço, caso contrario ela troca o ponto mais distante por ele se ele for mais próximo. Vale a pena observar que a fila de prioridade é implementada como um par da distância e do índice do ponto ao invés de um da distância e do propio ponto, isso ocorre, pois na segunda opção seria necessário criar um comparador ao invés de utilizar o ja pronto no std. Além disso, usamos o quadrado da distância euclidiana ao invés da distância euclidiana só por que isso mantêm a ordem e evita calcular a raiz quadrada(sendo uma operação custosa).

```
_querry (no, d):
    d = d%dim
    if no é uma folha then
        push(no.ponto)
        return
    else
        if p[d] < no.plano then
            galhoProximo = no.esq
            galhoOposto = no.dir
        else
            galhoProximo = no.dir
            galhoOposto = no.esq
        end
        _querry(galhoProximo, d + 1)
        if filaPrioridade.tamanho() < tamanhoDesejado ∨
          filaPrioridade.maiorDistancia() < |p[d] − no.plano| then
            _querry(galhoOposto, d + 1)
        end
        return
    end
```

A lógica da querry é a seguinte, se estiver em uma folha ela tenta colocar o ponto

na fila de prioridade, caso contrario ela calcula em qual dos dois galhos o ponto da entrada estaria se ele estivesse na árvore e o oposto, primeiro ela se chama recursivamente no primeiro, depois ela testa se o outro galho tem algum ponto que pode entrar na fila de prioridade, ou pela fila ainda ter espaço, ou por ela ainda poder ter um ponto que esta mais próximo do mais próximo encontrado ate agora. Esta comparação é aproximada olhando a distância do ponto para o plano, tem como fazer um algorítimo mais complicado para evitar outros casos considerando melhor a borda de onde os pontos podem estar, porem este algorítimo ou seria mais complicado, ou precisaria guardar mais informação no nó, logo eu decidi não implementá-lo..

2.4. indices dos nos

Em minha defese, a segtree normal implementada na programação competitiva usa uma ideia parecida em que o indice determina o indice dos filhos.

O índice de cada no é um par de inteiros (i, j), o significado deles e que todas as folhas do galho i e j estão nos índices entre i e j do vetor de nos externos, logo se $i = j$ isso significa que o no é uma folha e que a raiz é indexada por $(0, n-1)$(onde n é o número de folhas). Para garantir que o galho esquerdo tenha e mesma quantidade de folhas ou uma a mais do que o galho direito, calculamos $m = \lfloor (i+j)/2 \rfloor$ e definimos os filhos esquerdo e direito respectivamente como (i, m) e $(m+1, j)$. Uma das vantagens de implementar desta maneira é que desde que separemos sempre usando esta mesma regra, então não precisamos guardar os índices dos filhos, pois podemos calcular o índice dos filhos a partir do índice do pai. Agora o único problema é decidir onde guardamos o plano do no interno de índice (i, j), poderíamos fazer uma matriz para guardar cada índice em uma posição diferente, porem isso iria gastar uma quantidade quadrática de espaço, porem podemos observar que todo vértice interno tem um m diferente, isso ocorre, pois, se um não é descendente do outro então eles lidam com segmentos disjuntos, logo $i < i'$ e $j < j'$ logo $m < m'$, mas caso um seja descendente do outro então ou $i' < j' \leq m$(se ele é descendente pelo filho esquerdo) logo $m' < m$ ou $m + 1 \leq i' < j'$(se ele for descendente pelo filho direito) logo $m' > m$. Dessa maneira cada no interno tem um m independente e podemos usá-lo como índice do vetor de pontos internos sem ocorrer conflito.

3. Analise Complexidade

Nesta seção n é o número de pontos do conjunto de treino.

3.1. Tempo

Na construção da árvore kd, precisamos rodar um quickselect de complexidade $O(n)$(o pior caso é $O(n^2)$, mas a chance de isso acontecer é a mesma do quicksort rodar em $O(n^2)$, então não há muito problema, sem falar que o vetor é embaralhado) seguido por duas chamadas recursivas com metade do tamanho logo a relação de recorrência é:

$$T(n) = 2T(n/2) + O(n)$$

Logo pelo teorema mestre a construção tem complexidade $O(nlogn)$.

Eu não consegui analisar a complexidade da querry media da árvore. Mas eu consegui rodar em 2 a 5 minutos para bancos de dados com 7000 pontos.

3.2. Espaço

Precisamos de $O(n)$ espaço para os nos internos e externos e $O(logn)$ para a pilha de execução durante a recurção.

4. Metricas

Aqui colocamos as métricas das bases de dados com apenas 2 classes. Foi calculado as métricas para outras bases, mas elas ocupariam muitas colunas. A acurácia foi melhor que chutar aleatoriamente as classes, mas não sei julgar se foi boa. Um caso interessante ocorreu no ring .dat onde os espaço pode ser separado em duas áreas, um com mais pontos da classe 0 e poucos da classe 1 e outro apenas com pontos da classe 1, o resultado foi uma baixa revocação para a classe 0 e uma baixa precisão para a classe 1.

	acurácia	revocação-0	precisão-0	revocação-1	precisão-1
banana.dat	89.01	85.74	89.51	91.70	88.64
magic.dat	80.70	92.19	80.85	59.29	80.29
phoneme.dat	87.43	92.60	89.91	75.01	80.81
pima.dat	72.00	83.05	75.47	53.21	64.43
ring.dat	67.60	34.81	100.00	100.00	60.83
spambase.dat	80.22	84.35	83.16	73.89	75.56
twonorm.dat	96.82	97.17	96.53	96.47	97.12

5. Conclusão

Apesar do trabalho ter sido dificil de entender as especificações, foi interessante inventar uma nova maneira de guardar a estrutura de uma arvore em um vetor.