Universidade de Lisboa

Instituto Superior Técnico

Projeto de Algoritmos e Modelação Computacional

Classificador TAN e Aprendizagem de BNCs'

Professor Paulo Mateus

LMAC

Mariana Oliva 83640

Pedro Lopes 83650

Ricardo Quinteiro 83653

20 de Maio de 2018

1 Introdução

Este projeto tinha como objetivo desenvolver um classificador com base em redes de Bayes. A aprendizagem do classificador é feita utilizando o Algoritmo de Prim que encontra a árvore geradora máxima de um determinado grafo pesado. Os dados biomédicos para a construção deste grafo provêm do *UCI machine learning repository*.

Neste relatório vamos então abordar as alterações feitas à primeira entrega e as opções de implementação escolhidas.

2 Alterações à 1ª entrega

No que toca à 1ª entrega, alterámos toda a implementação do `WGraph` de modo a tornar a sua pesquisa mais eficiente e o código do Algoritmo de Prim mais simples. Apesar da implementação prévia do `WGraph` ocupar menos memória quando gravado, o Algoritmo de Prim tornava-se significativamente mais ineficiente. Deste modo, optámos por uma implementação do `WGraph` que permite aceder diretamente a todas as arestas que ligam a um nó, já que é necessário fazê-lo várias vezes quando se aplica o método `MST`.

Para além disso modificámos também a função `count` da amostra porque, devido a uma falha de interpretação nossa, esta não calculava o que era pedido. É de referir também que, por causa de uma distração, o Algoritmo de Prim enviado aquando da primeira entrega devolvia exatamente o oposto do que era pedido. Este encontrava a árvore geradora mínima ao invés da árvore geradora máxima.

3 Opções de Implementação

3.1 Amostra

No nosso projeto, a amostra (`Sample`) trata-se de uma LinkedList de vetores de inteiros e como tal a função `add`, `length` e `element` usam simplesmente o método `add(v)`, `size()` e o método `get(int index)`, respetivamente, da classe LinkedList.

Apesar de serem necessárias mais linhas de código, as funções `count` e `domain` também não apresentam dificuldade.

3.2 WGraph

A classe `WGraph` designa grafos pesados e implementa os métodos `add_edge`, `remove_edge` e `MST`, para além de ter obviamente um construtor da classe.

Nós optámos por definir os nossos grafos pesados como sendo um ArrayList (em que cada posição corresponde a um nó do grafo) e cada uma das suas entradas é uma LinkedList com `Edge`(s).

A classe `Edge` foi criada para auxiliar a classe `WGraph` e as suas propriedades são um `dvertex` (destination vertex) e um `weight`. Deste modo, quando se aplica o método `add_`-

`edge` adiciona-se uma `Edge` às LinkedLists correspondentes a cada nó da nova aresta para que este não seja um grafo dirigido e pesado ao mesmo tempo.

Para o método `MST` utilizámos o algoritmo de Prim para obtermos a árvore geradora maximal. Apesar deste algoritmo ser, geralmente, utilizado para encontrar a árvore geradora minimal, basta fazer duas pequenas alterações: atribuir no início o menor valor negativo ao custo de todos os nós (não é necessário fazê-lo para a raíz) e depois ir alterando os custos escolhendo o maior em vez do menor.

3.3 DGraph

Relativamente à classe DGraph definimos cada grafo dirigido como sendo uma ArrayList constituída por LinkedLists de inteiros em que cada posição da ArrayList corresponde a um nó. Cada entrada da ArrayList trata-se portanto de uma LinkedList cujos inteiros determinam os nós aos quais o nó correspondente à posição da ArrayList está ligado.

Assim sendo, os métodos `add_node()`, `add_edge(o,d)`, `remove_edge(o,d)` e `parents(n)` tornam-se bastante simples. Para o método `add_node()` basta incrementar uma unidade à dimensão do grafo e acrescentar uma LinkedList vazia no fim da ArrayList. Para adicionar (ou remover) uma aresta entre o e d, (`add_edge(o,d)` ou `remove_edge(o,d)`) basta ir à posição `o` na ArrayList e acrescentar (ou remover) o valor `d` à LinkedList. Por fim, o método `parents(n)` corre cada posição da ArrayList para verificar se a LinkedList de cada posição contém o nó `n`. Se contiver é porque o nó correspondente a essa posição se trata de um pai do nó `n`. Todas estas posições são guardadas numa LinkedList e devolvida no fim como resultado.

3.4 Construção do grafo pesado

A construção do grafo pesado é feita na App.java aquando do cálculo dos pesos:

$$I_T(X;Y|C) = \sum_{x,y,c} Pr_T(x,y,c)\, log\left(\frac{Pr_T(x,y,c)Pr_T(c)}{Pr_T(y,c)Pr_T(x,c)}\right)$$

Em que cada uma destas probabilidades usa os métodos `count()` e `length()` da classe `Sample` para contar o número de ocorrências de x, y ou c e dividir pelo tamanho da amostra.

Temos então que ter dois ciclos para variar os nós e ainda 3 ciclos para fazer o somatório de x, y, c. Assim sendo, para cada par (X_i, X_j) adicionamos então uma aresta entre estes dois nós com o respetivo peso $I_T(X_i; X_j|C)$.

3.5 Construção da rede de Bayes

Na rede de Bayes foi necessário guardar tanto o grafo como as probabilidades condicionadas. Existem três tipos diferentes de nós quando falamos de pais: a classe, que não tem pais; a raiz da árvore à qual a classe foi ligada e portanto tem um pai; e todas os restantes nós que têm dois pais. Guarda-se assim o grafo numa variável `G` de tipo `DGraph`,

e as probabilidades dos três tipos de nós em três variáveis diferentes. A probabilidade da classe é guardada num vetor de tipo *double*, denominado `Cprob`, uma vez que a única dimensão a considerar é o valor da classe. De maneira semelhante guarda-se a probabilidade condicionada da raiz em relação à classe numa matriz de tipo *double*, `Rprob`, pois desta vez temos que considerar duas dimensões (os valores da classe e da raiz). As restantes probabilidades são guardadas num ArrayList de tipo *double* em três dimensões em que na posição i do ArrayList encontram-se as probabilidades condicionadas para a variável $i + 1$ da amostra.

3.6 Aplicação para gerar a rede

A aplicação (App.java) para gerar a rede é a mesma aplicação que lê a amostra, cria o grafo pesado, aprende a rede de Bayes e grava-a no disco.

Usando o JFileChooser, a aplicação permite-nos escolher um ficheiro ao clicar no botão Browse. Este ficheiro .csv vai ser lido pela aplicação e os valores de cada linha vão ser inseridos em vetores de inteiros que vão sendo sucessivamente adicionados à variável `sample`, do tipo `Sample`, com que vamos trabalhar. Depois disso, o grafo pesado é construído (Secção 3.4) e o `MST` aplicado usando como raiz o nó 0. Como resultado obtemos um grafo dirigido ao qual teremos que acrescentar arestas entre a classe C e todos os nós antes de podermos gerar a rede. Para esta alteração bastou usar um ciclo para correr todos os nós e o método `add_edge` da classe `DGraph`. Temos agora tudo o que é preciso para gerar a rede de Bayes e gravá-la no disco. A rede vai ser guardada com diferentes nomes dependendo da amostra de que se trata para ser mais fácil para a App2.java ir buscar o ficheiro correspondente à doença que se quer diagnosticar.

3.7 Aplicação para classificar

Nesta aplicação (que se encontra no ficheiro App2.java) escolhe-se a doença a diagnosticar e introduz-se um vetor com os dados do paciente. Caso o input seja válido, esta devolve um output com o diagnóstico da doença (positivo ou negativo). Caso contrário, devolve uma mensagem de erro.

Ao receber o input, são criados dois vetores iguais ao recebido com o valor da classe acrescentado. Um deles tem o valor 1 e o outro 0. De seguida é calculada a probabilidade (utilizando o método `prob` da classe `BN`) de observar esse vetor no domínio da rede de Bayes criada. Após esse cálculo, verifica-se qual é o valor mais elevado. Se for o que tem a classe com valor 1 então o diagnóstico é positivo, caso contrário é negativo.