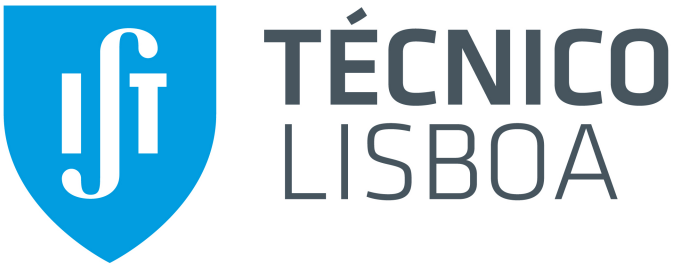

Instituto Superior Técnico

Mestrado Integrado Em Engenharia Eletrotécnica e de Computadores

Programação Orientada a Objetos

Relatório de Projeto

Trabalho realizado por:	*Número:*
Diogo Moura	86976
Diogo Alves	86980
Luís Crespo	87057

Grupo 7

2019/2020

Conteúdo

1 Introdução

O objetivo deste trabalho é construir um classificador de Bayes (*Bayesan Network Classifier*) que consiga operar num dado *dataset* utilizando interfaces e classes que possam ser extensíveis, colocando em prática os nossos conhecimentos sobre a linguagem java e modelação UML. Assim tentámos usar, na medida do possível, código coerente e bem estruturado usando características especiais da linguagem *Java* como *exceptions*, *classes* e *interfaces* fazendo uso do paradigma da linguagem orientada a objetos.

2 Extensibilidade

De forma geral, consideramos que a extensibilidade foi implementada de forma correta no nosso código. Tentámos utilizar interfaces sempre que possível, definindo nestas os serviços (métodos) a implementar nas classes abaixo. Em algumas situações usámos ainda classes abstractas entre a interface e as classes concretas. As classes/interfaces foram também separadas em pacotes que conseguem operar de forma relativamente independente (embora as classes do pacote *classifier* utilize classes de todos os outro pacotes). O uso de tipos genéricos nas classes dos pacotes *directedTree* e *graph* contribui também para o aumento da extensibilidade do código. Existe também a implementação da interface *Iterable* do *java.lang* numa das nossas interfaces - *Dataset* - (e o respetivo *Iterator*), de forma a que seja mais fácil percorrer o *Dataset* (através de um *enhanced for loop* por exemplo).

3 Análise crítica de performance e estruturas de dados utilizadas

3.1 Estruturas de dados

Para guardar em memória a associação correspondente ao *Attribute* e o repetivo valor (*Integer)* na *DefaultInstance* foi utilizado um *HashMap*. Um *Map* é uma estrutura de dados que permite guardar pares de chave-valor (*key-value pairs*). No nosso caso as *keys* correspondem ao atributo e os *values* ao respetivo valor naquela amostra. Foi escolhido o *HashMap* por este ter uma complexidade $O(1)$ na procura e na inserção.

A complexidade de inserção de um novo vértice no grafo à lista dos vértices é $O(1)$ e a de adicionar uma nova ligação à matriz de adjacências do grafo é $O(1)$, pelo que a criação de um grafo completo com V vértices e E arestas é $O(|V|+|E|)$. Estas complexidades sãos as melhores possíveis de obter num grafo. No nosso caso específico, como todos os vértices estão ligados uns aos outros o número de vértices é dado por $O(V^2)$ e, assim, a complexidade de criação do grafo é também $O(V^2)$.

De forma a construir uma *directed tree* o mais equilibrada possível para otimizar a procura na árvore,

implementou-se um método que indica a melhor *root*. Este método tem uma complexidade de $O(N^2)$ e permite que cada procura tenha no máximo uma complexidade, para os exemplos fornecidos, $O(N/2)$ em vez de $O(N)$. Os métodos implementados de procura na árvore são todos com base em procura em profundidade devido às características das árvores obtidas.

3.2 Algoritmo de Prim

A complexidade do algoritmo de Prim num grafo com V vértices, utilizando uma matriz de adjacências, como no nosso caso, é $O(V^2)$. Caso utilizássemos uma árvore binária (*binary heap*) para guardar as arestas do grafo, ordenadas pelo seu peso, a complexidade temporal seria de $O((|V|+|E|)log|V|)$, ou seja, $O(|E|log|V|)$, sendo E igual a V^2, equivaleria a $O(V^2log|V|)$, o que levaria a uma pioria da performance, pelo que não optámos por essa implementação. Seria também possível utilizar uma árvore de Fibonnaci (*Fibonnaci Heap*), com complexidade $O(|E|+|V|log|V|)$, o que equivale a $O(V^2)$, para $E = V^2$, isto é para todos os vértices ligados entre si. Assim, o método por nós utilizado foi o mais simples e o melhor em termos de performance, já que todos os vértices se encontram ligados entre si.

3.3 Algoritmo de Prim vs Kruskal

O algoritmo de Prim é melhor que o de Kruskal para o nosso caso, já que o primeiro apresenta complexidade O(V^2) e o segundo O(|E|log|E|), o que uma vez mais, para o caso em que os vértices estão todos ligados uns aos outros ($E = V^2$), levaria a uma complexidade maior equivalente a $O(|E|log|E|) = O(V^2logV^2) = O(2V^2logV) = O(V^2logV)$.

3.4 Parâmetros do Classificador

Na nossa solução, as contagens N_{ijkc} e os parâmetros θ_{ijkc} não são guardados numa estrutura de dados. São sim calculados quando são necessários. Os métodos que calculam N_{ijkc} e θ_{ijkc} têm complexidade $O(N*k)$, em que k é o número de atributos e N o número de amostras do conjunto de treino. Assim, para calcular a probabilidade de uma amostra de teste pertencer a uma classe, a complexidade é $O(N * k^2)$ e portanto classificar uma amostra tem complexidade $O(N * k^2 * s)$, sendo s o número de classes. Em conjuntos de dados suficientemente grandes, esta complexidade poderia ser reduzida, à custa de ser aumentada a memória utilizada, se os valores de N_{ijkc} e θ_{ijkc} forem guardados numa estrutura de dados à medida que fossem sendo necessários calcular, e da próxima vez que fossem necessários já não seria preciso voltar a calcular. Outra solução seria calcular e guardar todos estes parâmetros *à priori* (durante a construção do classificador). Desta forma, o tempo de classificação (*time to test*) iria diminuir mas o tempo de construção (*time to build*)

iria aumentar.

Já o cálculo dos parâmetros α_{ij} tem complexidade $O(N * k * s * r^2)$, sendo r o número médio de valores que um atributo pode assumir. Assim, como é necessário invocar esta função $k^2/2$ vezes, o que corresponde ao número de arestas do grafo, a complexidade de construir o grafo é de $O(N * k^3 * s * r^2)$.

4 Conclusões

Os objetivos foram atingidos já que conseguimos produzir classes extensíveis e fazer uso do polimorfismo. Usámos também funcionalidades próprias da linguagem de forma ampla e sempre que possível, nomeadamente *exceptions* e objetos das classes já incorporadas na linguagem, ou seja dos pacotes *java.util* e *java.lang*.

A solução apresentada não contém qualquer erro ou *warning*.

Os resultados obtidos em termos de *performance* e métricas do classificador são exatamente iguais aos desejados, apresentados pela professora.