Princípios de Programação

Trabalho Sexto

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

2020/2021

Estima-se que no ano de 2018, a fraude com cartões de crédito tenha atingido 20 mil milhões de euros em todo o mundo. Uma grande parte destes casos são feitos a partir de dados de cartões de crédito extraídos de lojas online que sofrem ciber-ataques. Serviços como o MBNet e Privacy.com previnem contra este tipo de ataques, usando cartões de crédito temporários, com limite temporal e restrições de valor e de vendedor. No entanto, a grande maioria dos utilizadores de cartões de crédito online não utiliza estes mecanismos.

Um método de deteção de fraude compara cada nova transação num dado cartão com um histórico de transações para o mesmo cartão. Neste trabalho vamos desenvolver um método para construir *métricas* para um dado cartão a partir de dados de transações. Um exemplo de uma métrica é o valor médio das transações de um dado mês do cartão de crédito. Outro exemplo é o somatório de todas as transações do cartão de crédito. De um modo geral, uma métrica é um valor que nos dá informação agregada sobre um histórico de transações passadas, agrupadas por certas características. Estas características temporais são úteis para detectar fraude, por exemplo usando algoritmos de Aprendizagem Automática, ou recorrendo a regras definidas pela empresa emissora dos cartões.

As métricas geradas pela nossa solução irão ser usadas para detectar fraude, mas esta utilização está fora do âmbito deste trabalho. Como exemplo de utilização da nossa aplicação poderiamos implementar uma regra que comunica ao Banco de Portugal todas as contas que façam transações que totalizem 100 000 euros no mesmo dia.

A nossa solução é responsável por receber uma sequência de transações (representadas por uma tabela em formato *Comma-Separated Values*, CSV) e de gerar novas colunas para essa tabela, colunas estas que representam os valores de umas dadas métricas para cada transação. Esta capacidade permite

usar o nosso sistema em tempo real para classificar cada transação à medida que a recebemos: de cada vez que chega uma nova transação geramos logo a linha correspondente.

Para permitir aos utilizadores do sistema configurar as métricas que melhor entenderem, vamos implementar uma linguagem de métricas muito simples, inspirada pelo SQL e que trabalham sobre as colunas de uma tabela. As colunas são numeradas a partir do zero: `0`, `1`, ...

A nossa solução está dividida em dois componentes: o `Agregador` e o `Processador`. O `Agregador` é uma ferramenta que permite construir uma dada métrica sobre uma sequência de transações. O `Processador` é responsável por processar uma lista de transações usando diferentes métricas, criando uma tabela de informação agregada.

A. Agregador O `Agregador` é um programa Haskell incluido no ficheiro `Agregador/Main.hs`. Recebe no *stdin* a descrição de uma métrica e, a partir desse ponto, recebe uma sequência de transações. Para cada nova transação emite a métrica calculada a partir de todas as transações recebidas anteriormente.

Como exemplo simples, uma instância do `Agregador` poderia utilizar a métrica `sum 1` para calcular a soma da segunda coluna de todas as transações recebidas até ao momento. As linhas que começam por > representam *stdin* e as que começam por < representam *stdout*. As marcas >␣ e <␣ nos exemplos não fazem parte da interação.

```
  $ cd Agregador && ghc --make Main.hs && cd ..
  $ ./Agregador/Main
> sum 1
> 0 1
< 1.0
> 0 2
< 3.0
> 0 5
< 8.0
> exit
```

O utilizador começa por definir a métrica (`sum 1`) e depois introduz a primeira transação (`0 1`). Como só foi vista uma transação, a soma é o valor da segunda coluna da transação em questão. Aquando da segunda transação (`0 2`) o valor total dessa coluna até ao momento é `3.0`. Depois da terceira transação esse valor é já `8.0`. No final, podemos terminar o programa introduzindo `exit`.

O `Agregador` suporta três tipos de métricas, a saber, `sum c`, `average c` e `maximum c`, que representam respectivamente a soma, média e máximo da coluna número `c` (com indíce a começar em zero). Após cada uma destas

métricas simples pode aparecer uma ou mais ocorrências do sufixo `groupby c`, onde `c` indica novamente uma coluna. Este sufixo irá restringir a métrica apenas às transações passadas que coincidem com a transação actual nas colunas a agrupar. Eis um exemplo:

```
> sum 1 groupby 2
> 0 100 3
< 100.0
> 0 20 3
< 120.0
> 0 40 3
< 160.0
> 0 2 25
< 2.0
> 0 7 25
< 9.0
> 10 10 3
< 170.0
> exit
```

Neste exemplo fazemos a soma para todas as combinações da terceira coluna. As três primeiras transações funcionam de forma semelhante ao exemplo anterior porque o valor da terceira coluna mantém-se inalterado. Mal se muda o valor da terceira coluna, recebemos o valor da segunda coluna da transação, isto é `2.0`. A transação seguinte volta a conter o valor `25` na terceira coluna, pelo que o `Agregador` devolve a soma de todas as transações analisadas anteriormente com valor `25` nessa coluna. Finalmente, se volvermos ao valor `3` para a terceira coluna, o `Agregador` devolve a soma das transações passadas com o valor `3` na terceira coluna.

O `Agregador` aceita vários agrupamentos, como por exemplo, `average 3 groupby 7 groupby 8 groupby 9`, de modo a calcular a média da quarta coluna de todas as transações analisadas no passado que tenham a oitava, a nona e décima colunas iguais à actual. Eis mais um exemplo, desta vez utilizando dois agrupamentos.

```
> average 1 groupby 2 groupby 3
> 0 4 1 4
< 4.0
> 1 12 1 4
< 8.0
> 2 8 1 4
< 8.0
> 3 54 2 4
< 54.0
> 4 1 2 2
```

```
< 1.0
> 5 7 2 2
< 4.0
> 6 0 2 4
< 27.0
> 7 32 2 2
< 13.333333
> exit
```

B. Processador O segundo componente do nosso sistema é um programa Java que lê um ficheiro CSV (`dataset.csv` no exemplo abaixo) e cria vários agregadores, um para cada métrica. As métricas são lidas de um outro ficheiro de texto (`metricas.txt` no exemplo). Para cada linha no ficheiro CSV, o `Processador` substitui as vírgulas por espaços e passa a linha assim obtida a cada um dos agregadores. Depois lê o resultado de cada agregador e escreve-o no *stdout*, separando os vários resultados por vírgulas. No exemplo abaixo, o `Processador` recebe um ficheiro de texto com seis métricas, pelo que o output obtido corresponde a uma tabela com seis colunas.

```
  $ cd Processador && javac src/*.java -d bin && cd ..
  $ java -cp Processador/bin Main metricas.txt dataset.csv Agregador/Main
< 435.0,435.0,435.0,435.0,435.0,435.0
< 4314.0,2157.0,2157.0,3879.0,3879.0,3879.0
< 8257.0,2752.3333,2752.3333,3943.0,3943.0,3943.0
  ...
```

O `Processador` recebe três argumentos: um ficheiro que define uma métrica em cada linha (cada métrica corresponde a uma coluna do output), um ficheiro CSV com as transações e o caminho para o binário do `Agregador`. Este programa deverá criar um agregador por cada métrica (usando a classe Java `ProcessWrapper` fornecida).

C. QuickCheck O `Agregador` deve conter três testes que considere relevantes. Os testes são exercitados quando se usa a *flag* `-t` na linha de comandos.

```
  $ ./Agregador/Main -t
< +++ OK, passed 100 tests.
< +++ OK, passed 100 tests.
< +++ OK, passed 100 tests.
```

D. Raciocínio sobre programas Mostre que, para toda a lista finita não vazia de floats *xs*, o máximo da lista xs é igual ao máximo dessa mesma lista pela

ordem inversa. Deverá começar por escrever as definições recursivas das funções relevantes. Poderá ainda usar (sem necessitar de provar) a lei

```
maximum (ps ++ qs) = maximum ps `max` maximum qs
```

Pode escrever a sua solução à mão ou num programa de texto, mas deverá entregar apenas um documento pdf.

Notas

1. O seu trabalho é constituído por um módulo Haskell `Main` na pasta `Agregador`, um projecto Java na pasta `Processador` com a classe `Main` e um documento pdf com a solução da parte D. Deverá submeter um ficheiro zip com o nome `t6_fcXXXXX_fcYYYYY.zip`, onde `XXXXX` e `YYYYY` são os vossos número de aluno, por ordem ascendente. Este ficheiro zip deverá conter as seguintes pastas e ficheiros:

   - `Agregador/Main.hs`
   - Outros ficheiros Haskell na pasta `Agregador`
   - `Processador/src/Main.java`
   - Outros ficheiros Java na pasta `Processador/src`
   - `raciocinio.pdf`

2. Para os valores constantes nas transações utilize o tipo de dados `Float` em Haskell e `float` em Java. Considere como significativas apenas as primeiras duas casas decimais de um número em vírgula flutuante. Em particular `21,798` deve ser considerado igual a `21,71285`.

3. Para tornar o *input* do `Agregador` síncrono com a linha de comandos, pode começar o programa com:

   ```
   import System.IO

   main :: IO ()
   main = do
     hSetBuffering stdin NoBuffering
     hSetBuffering stdout NoBuffering
     ...
   ```

4. Cada função (ou expressão) que escrever em Haskell deverá vir sempre acompanhada de uma assinatura. Isto é válido para as funções enunciadas acima bem como para outras funções ajudantes que decidir implementar.

5. Para tratar conjuntos de ficheiros e linhas de ficheiros no `Processador` é exigida a utilização de *streams* e funções de ordem superior. O método `java.nio.file.Files.lines(Path)` poderá set útil neste processo.

6. O código Java deve estar documentado no formato *javadoc*.
7. Os trabalhos serão avaliados automaticamente. Respeite os nomes dos ficheiros e uso esperado.
8. Lembre-se que as boas práticas de programação Haskell e Java apontam para a utilização de várias funções ou métodos simples em lugar de uma função ou método único mas complicado.

Entrega. Este é um trabalho de resolução em pares. Os trabalhos devem ser entregues no Moodle até às 23:55 do dia 11 de janeiro de 2021.

Plágio. Os trabalhos de todos os alunos serão comparados por uma aplicação computacional. Relembramos aqui um excerto da sinopse: "Alunos detetados em situação de fraude ou plágio, plagiadores e plagiados, ficam reprovados à disciplina (sem prejuízo de ser acionado processo disciplinar concomitante)".