Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 1

Relatório de desenvolvimento

João Correia
(A84414)

Marco Pereira
(A89556)

4 de abril de 2021

Resumo

O presente relatório tem como objetivo descrever o processo de desenvolvimento da ferramenta *CSV2JSON*, um algoritmo de conversão de ficheiros CSV em ficheiros no formato JavaScript Object Notation (JSON).

O trabalho desenvolvido concretizou-se numa ferramenta capaz de converter ficheiro CSV que utilizem diversos delineadores de campos, suportando, também, um conjunto diverso de operações sobre os dados contidos no ficheiro.

Embora o escopo da ferramenta seja amplamente maior do que o originalmente proposto, o que resulta num sistema de complexidade considerável, este retém, no entanto, uma elevada eficácia de processamento.

Conteúdo

Capítulo 1

Introdução

1.1 CSV2JSON

Área: Processamento de Linguagens

O seguinte projeto encontra-se inserido no contexto da unidade curricular de Processamentos de Linguagens, presente no 3º ano letivo do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

Este constitui o primeiro trabalho prático presente na componente prática da avaliação da unidade curricular, cujo objetivos avaliativos se centram na capacidade de desenvolvimento de *Expressões Regulares (ER)* de descrição de padrões textuais e no desenvolvimento de sistemas capazes de transformar texto numa lógica *condição-ação*. Este tem, como suporte, a linguagem de programação Python, utilizando o módulo *re* [1], que disponibiliza diversas ferramentas de reconhecimento e transformação textual através do uso de expressões regulares.

O tema a abordar será o desenvolvimento de uma ferramenta capaz de converter ficheiros CSV numa versão equivalente no formato JavaScript Object Notation (JSON). O ficheiro original, poderá, contudo, conter operadores especiais que indiquem a necessidade de processar os dados associados ao operador antes da conversão destes para JSON. Estas operações serão descritas em secções futuras do presente relatório.

Este relatório terá como objetivo a descrição detalhada dos requisitos do projeto, assim como o de detalhar o processo de concepção e implementação do algoritmo responsável pela sua conversão, sublinhando as suas funcionalidades, tanto as especificadas como requeridas no enunciado do projeto, como as funcionalidades adicionais reconhecidas como úteis por parte do grupo responsável pelo desenvolvimento da ferramenta.

A ferramenta desenvolvida revela-se eficaz na tarefa proposta, contendo as funcionalidades necessárias para servir como um conversor útil em tarefas do mundo real onde se veja a necessidade de conversão e processamento de dados presentes em ficheiros CSV.

Estrutura do Relatório

O presente relatório encontra-se dividido em 5 secções. A primeira secção, afigura-se como a introdução ao trabalho, secção onde a atual descrição da estrutura do relatório se encontra presente.

No segundo capítulo, 2, o projeto proposto é analisado, sendo levantado um conjunto de requisitos, tanto mencionadas no enunciado deste como também requisitos adicionais inferidos como úteis pela equipa responsável pelo projeto.

O terceiro capítulo, 3, irá descrever a forma como foi desenvolvida a ferramenta, indicando os vários passos de leitura e processamento dos ficheiros, até à sua transformação total em ficheiros no formato JSON.

No capítulo 4, o processamento descrito no capítulo anterior é implementado sob a forma de um programa desenvolvido na linguagem de programação Python. Aqui são apresentados os detalhes técnicos do funcionamento da ferramenta.

Por fim, o quinto capítulo, 5, conclui o relatório, sintetizando a informação disponibilizada ao longo deste.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O sistema a desenvolver deverá ser um conversor de ficheiros gravados em formato *Comma separated Values* (CSV) para ficheiros em formato *JavaScript Object Notation* (JSON), um formato de ficheiro chave ⇒ valor. O cabeçalho do ficheiro CSV descreverá os campos que servirão como chaves no correspondente ficheiro JSON criado.

O cabeçalho do ficheiron JSON suportará ainda operações extra, a presença do carácter $*$ no final do nome de um campo indicará que este é uma lista de valores a ser representados dessa forma no ficheiro JSON. É ainda possível especificar operações a calcular sobre essa lista de valores caso essa seja indicada após o $*$. Cada linha consequente do ficheiro CSV indicará o conjunto de valores correspondentes a cada chave indicada no cabeçalho.

Um pequeno exemplo ilustrativo:

```
nome;numeros*;notas*avg
Ana Maria;(845,5900);(12,14,15,18)
João Martins;(850;6000);(11,16.5,16)
```

Neste CSV o cabeçalho indica a presença de 3 campos diferentes: o campo nome, o campo números, que será uma lista de valores e o campo notas que será uma lista de valores dos quais irá calculará a média.

O correspondente ficheiro JSON será:

```
[
  {
    "nome": "Ana Maria",
    "numeros": [845,5900],
    "notas_avg": 14.54
  },
  {
    "nome": "João Martins",
    "numeros": [850,6000],
    "notas_avg": 14.5
  }
]
```

2.2 Especificação do Requisitos

Do enunciado referido na secção anterior é possível levantar um conjunto de requisitos que o projeto deverá cumprir. O grupo, no entanto, julgou-os insuficientes para o desenvolvimento de uma ferramenta completa, pelo que foi levantado um conjunto de requisitos adicionais que tem o objetivo de complementar o funcionamento da ferramenta.

2.2.1 Requisitos base

Os requisitos base são os que é possível extrair do enunciado presente na secção 2.1. Estes são, de seguida, enunciados.

1. Separar os vários campos do cabeçalho do ficheiro através do carácter ';';
2. Identificar quais campos possuem o carácter *;
3. Dos campos que possuem o carácter *, identificar quais possuem operações a aplicar sobre os valores
4. Em cada linha do corpo do CSV, separar os vários campos através do carácter ';';
5. Nos campos de uma linha afetados pelo operador *, separar os vários valores que se encontram separados pelo carácter ',';
6. Aplicar as operações definidas no cabeçalho à lista de valores, caso existam;
7. Converter os valores lidos para o correspondente par chave $\Rightarrow$ valor, conjundando a chave lida no cabeçalho com o valor lido na linha.

2.2.2 Requisitos adicionais

Embora os requisitos acima mencionados constituam uma base sólida para um conversor, um conjunto de funcionalidades necessárias para que este seja uma ferramenta completa encontram-se em falta. Como tal, o grupo decidiu inclui-los no levantamento de requisitos. Estes são:

1. Conseguir também processar ficheiros CSV cujos campos sejam separados por vírgulas, visto este ser o formato mais comum de ficheiros CSV;
2. Suportar campos em falta em cada linha do ficheiro;
3. Quando o separador de campos é a vírgula, suportar o carácter ';' como separador de valores em campos que contém listas numéricas;
4. Permitir várias operações sobre a mesma lista de valores, incluindo um operador que permite manter o conjunto de valores quando se aplicam outras operações, funcionalidade originalmente não presente, pois, quando se aplicava uma operação a lista de valores original era eliminada do ficheiro JSON;
5. Permitir converter valores lidos em valores numéricos, pois, por base, todos os valores não presentes em listas são convertidos para string. Esta funcionalidade é ativada utilizando o carácter + à frente do nome do campo no cabeçalho, à semelhança do operador *;
6. Permitir escolher o nome do ficheiro de input e output;
7. Introdução de mecanismos de notificação de irregularidades nos ficheiros.

Segue um exemplo que encapsula todas as novas funcionalidades:

python csv2json.py teste.csv conversao.json

Ficheiro Input: teste.csv

```
Nome,Id,Numero+,Notas*group;avg;max
João,1,123,(2;4)
Ana,2,,(5;1)
Marco,3,,(9;8)
```

Ficheiro Output: conversao.json

```
[
  {
    "Nome": "João",
    "Id": "1",
    "numero": 123.0,
    "Notas": [2.0, 4.0],
    "Notas_avg": 3.0,
    "Notas_max": 4.0
  },
  {
    "Nome": "Ana",
    "Id": "2",
    "Notas": [5.0, 1.0],
    "Notas_avg": 3.0,
    "Notas_max": 5.0
  },
  {
    "Nome": "Marco",
    "Id": "3",
    "Notas": [9.0, 8.0],
    "Notas_avg": 8.5,
    "Notas_max": 9.0
  }
]
```

Capítulo 3

Concepção/desenho da Resolução

Devido à natureza dos ficheiros CSV, o processamento linha a linha destes surge como a estratégia mais razoável, visto não haver dependência entre os valores das várias linhas do ficheiro, com exceção da relação entre o cabeçalho e o corpo.
Como tal, será necessário realizar, em primeiro lugar, um processamento do cabeçalho do ficheiro CSV de forma a extrair a informação necessária ao processamento do resto do ficheiro.

3.1 Processamento do cabeçalho

O cabeçalho do ficheiro CSV contém informação sobre cada campo presente no ficheiro. Para além do nome do campo, poderá conter tanto o operador de grupo (*) como o operador de *cast* para valor numérico (+). Caso contenha o operador *, pode ainda conter operações a aplicar ao conjunto de valores.
Como tal, o fluxo de processamento do header deverá seguir os seguintes passos:

1. Inferir se o separador de campos é a vírgula ou o ponto e vírgula;
2. A partir do separador de campos, obter o separador de operações dentro do mesmo campo (caso o separador de campos seja vírgula, o separador de valores será o ponto e vírgula e vice-versa);
3. Separar cada campo do cabeçalho do ficheiro, capturando 3 grupos por campo: o nome do campo, a presença do operador + ou * (opcional), e operações sobre os valores (opcional);
4. Guardar, numa estrutura de dados, que operações devem ser aplicadas a cada campo;
    i) Caso apenas esteja presente o nome do campo, é registada a operação de cópia;
    ii) Caso esteja presente o operador +, é registada a operação de *cast* para valor numérico;
    iii) Caso seja detetado o operador * sem nenhuma operação à frente, é registada a operação de listagem;
    iv) Caso sejam detetadas operações associadas ao operador *, guardar cada uma das operações, separando-as pelo carácter separador de operações;
5. Devolver o carácter separador de campos, o carácter separador de operações, o nome de cada campo e as operações associadas a cada campo.

3.2 Processamento do corpo

Para além do corpo do CSV, a componente da ferramenta que seja responsável pelo processamento do corpo necessitará de receber os quatro dados mencionados no final da secção anterior: o carácter separador de campos, o carácter separador de operações, o nome de cada campo e as operações associadas a cada campo. O corpo do CSV será processado como um conjunto de linhas que serão iteradas uma a uma.
O processo de transformação segue os seguintes passos:

1. Criar uma lista onde todas as linhas do ficheiro JSON serão inseridas;
2. Inserir na lista o delimitador inicial do ficheiro JSON: o carácter '[';
3. Iterar sobre cada linha do corpo;
   3.1) Inserir na lista o carácter '{';
   3.2) Separar os campos da linha utilizando o carácter separador de campos;
   3.3) Iterar sobre cada campo presente na linha;
      3.3.1) A cada campo associar o correspondente nome da lista de nomes de campos fornecida, assim como o conjunto de operações a aplicar a esse campo.
         i) Caso a operação associada seja cópia, inserir a linha **"chave": "valor"** na lista;
         ii) Caso a operação associada seja *cast*, inserir a linha **"chave": valor** na lista;
         iii) Caso contrário, significa que se está a tratar de um campo que contém uma lista de valores, como tal, separam-se os vários valores existentes no campo. De seguida aplica-se cada uma das operações associadas ao campo, inserindo, para cada uma, a linha **"chave_operação": valor_calculado**;
   3.4) Inserir na lista o carácter '}', seguido de vírgula caso não se trate da última linha do ficheiro;
4. Inserir na lista o delimitador final do ficheiro JSON: o carácter ']';
5. Converter todas as strings presentes na lista numa única, unindo-as pelo carácter de newline '\n', formando, assim, uma string em formato JSON correto.

Capítulo 4

Codificação e Testes

A ferramenta foi desenvolvida utilizando a linguagem de programação Python, recorrendo à biblioteca RE de forma a possibilitar a utilização de expressões regulares no processamento textual.
O código encontra-se dividido em 3 blocos lógicos:

1. Processamento do cabeçalho;
2. Processamento do corpo;
3. Lógica de leitura e escrita de ficheiros.

4.1 Processamento do cabeçalho

O algoritmo de processamento do cabeçalho encontra-se descrito na secção 3.1. A sua implementação encontra-se disponível no anexo A e, de forma a facilitar a leitura, encontra-se, no anexo B o código sob a forma de imagem. Um pormenor a ter em consideração na interpretação do código do primeiro anexo é o valor **placeholder_for_EOL** que deve ser interpretado como o carácter $, porém teve de ser alterado por um placeholder devido à sintaxe do latex.
Os detalhes de assinatura da função encontram-se descritos na sua **docstring**, associada também à funcionalidade de *type hinting*, presente na linha 6.
As operações permitidas sobre grupos de valores encontram-se descritas na linha 24, estas são:

- **sum**: Somar todos os valores na lista;
- **avg**: Calcular a média de valores na lista;
- **max**: Indicar o maior valor presente na lista;
- **menor**: Indicar o menor valor presente na lista;
- **group**: Copiar os valores da lista para o ficheiro JSON (Na versão do enunciado, quando é aplicada uma operação sobre os valores, os valores originais não eram copiados, perdendo-se informação).

De forma a inferir qual o separador de campos presente no texto usa-se a seguinte expressão regular:

```
[^+*;,]+(?:\*|\+)?(?:[^;,]+)?(;|,|$)
```

Sabendo que um campo do cabeçalho tomará sempre uma de quatro formas: **campo**, **campo***, **campo+** ,ou **campo*(conjunto de operadores)**, é possível capturar o carácter que está a ser usado para separação de campos, visto que este surgirá no final do nome do campo, do operador *, ou no final do conjunto de operadores da lista de valores.

Para tal foi usada a função *match* e utilizou-se o método *group(1)* para aceder ao grupo de captura que contém o carácter de separação de campos.

Tendo sido capturado o carácter de separação de campos é possível inferir o carácter de separação de operações pois será o seu contrário na lógica de delimitadores, isto é: se o carácter de separação de campos for ';' o de operações será ',' e vice-versa.

De forma a extrair os vários campos é usada a expressão regular:

```
([^+*;]+)(\*|\+)?([^;]+)?
```

Esta expressão é utilizada na linha 29, quando se confirma que o carácter de separação de campos é o ';', caso o cáracter seja ',', é usada uma expressão equivalente mas utilizando ',' ao invés de ';' (ver linha 33).

Esta expressão regular contém 3 campos de captura distintos:

- Captura do nome do campo (obrigatório);
- Captura do operador de grupo ou cast (opcional);
- Captura das operações a aplicar ao grupo de valores (opcional e dependente do segundo grupo).

Utilizando a função **findall** do módulo *re*, é possível obter uma lista de tuplos de 3 valores, cada tuplo contendo a informação acima mencionada sobre cada campo.

Desta forma, o bloco de código que inicia na linha 35, inicia essa iteração sobre os vários tuplos, guardando em duas listas separadas o nome dos campos e as operações a aplicar a cada campo.

4.2 Processamento do corpo

O algoritmo de processamento do corpo do CSV encontra-se descrito na secção 3.2. A sua implementação encontra-se disponível no anexo C. No entanto, devido à mecânica de identação por bloco do Python, a sua legibilidade no anexo não é satisfatória, pelo que se aconselha a consulta das funções que compõe o código fonte na forma de imagens no anexo D.

A função principal deste bloco receberá a lista de linhas que compõe o corpo do CSV, assim como o carácter delimitador de campos e o delimitador de operações, a lista de nomes dos campos e a lista de operações a aplicar a cada campo.

É criada uma lista inicial onde todas as strings correspondentes a linhas do ficheiro JSON serão inseridas.

Cada linha do corpo será iterada e processada. De forma a separar cada campo que compões uma linha do CSV, é usada a função *split*, usando o carácter delimitador de campos como critério de separação.

Tendo a lista de campos que compõe essa linha, é iterada ao mesmo que é iterada a lista com os nomes dos campos e a lista com as operações, de forma a ir construíndo o bloco JSON correspondente a essa linha do CSV.

Em casos de campos que contenham uma lista de valores, é necessário retirar os valores que se encontram dentro dos parentesis (ex: dado o campo "(1,4,7.8)"é necessário extrair a string "1,4,7.8". Para tal, é usada a seguinte expressão regular:

```
\((.+?)\)
```

Esta expressão utiliza o operador de captura *lazy* para capturar todo o texto entre dois parêntesis.

Tendo o texto entre os parêntesis, este é separado tendo como critério de separação o carácter de separação de operações (e não o delimitador de campos). Estes valores são depois dados como argumento à função **process_operations** de forma a aplicar as várias operações que o cabeçalho do ficheiro indica que devem ser aplicadas a esse campo.

Por fim, todas as strings exisxtentes na lista são concatenadas usando o carácter newline '\n', formando uma string correspondente ao ficheiro JSON.

4.3 Código de leitura e processamento de input do utilizador

Este bloco de código, presente no anexo E representa a interação do programa com o utilizador, lendo o seu input e tratando-o.

O utilizador pode escolher o ficheiro de input, desde que este esteja presente na pasta *input* e escolher ainda o nome do ficheiro de output, que será criado na pasta *output*. Caso estes valores não sejam fornecidos, os ficheiros data.csv e data.json são escolhidos, por definição.

O ficheiro de input é aberto e as suas linhas separadas utilizando a função *read* seguida da função *splitlines*. Esta combinação de funções tem um resultado diferente que utilizar a função *readlines* pois a primeira opção remove os caracteres de newline no final de cada linha, ao invés da segunda, que os mantém.

As linhas são processadas usando o código apresentado nas duas secções anteriores, sendo, por fim, aberto um novo ficheiro onde a string resultante do processamento é escrita, criando o ficheiro JSON pretendido.

O tempo de processamento do ficheiro é medido e apresentado ao utilizador.

4.4 Alternativas, Decisões, Problemas de Implementação e limitações

Uma decisão relevante que teve um elevado impacto no tempo de processamento da ferramenta é a utilização de uma lista para guardar todas as string resultantes do processamento do corpo do csv, ao invés de concatenar sucessivamente os resultados numa única string.

Isto deve-se ao facto da função *append*, que permite inserir um elemento no final de uma lista de python, tem complexidade temporal constante, isto é, $O(1)$. No caso de concatenação de strings, como todos os caracteres de ambas são copiados para uma nova string, a complexidade é $O(N + M)$, em que N e M são os comprimentos das respetivas strings, o que resulta num tempo de execução significativamente superior.

Por fim, utilizando a função *join* é possível juntar todas as strings presentes na lista numa única, reduzindo ao máximo o número de concatenações presentes no programa.

Quanto a alternativas de programação, a mais relevante centra-se na metodologia de separação dos campos do corpo do csv, presente na linha 31 do anexo C. Inicialmente, ao invés da função *split*, era usada a seguinte linha de código:

```
fields = re.findall(rf'([^{field_delimiter}]+)(?:{field_delimiter}|$)|;', line)
```

A utilização desta expressão regular configurava-se ideal em quase todos os cenários, excepto num: no caso do último campo da linha não se encontrar preenchido, isto é, para o csv:

```
nome;numero
Ana Maria;8
;9
João Martins;
```

A expressão é capaz de identificar 2 campos separados na segunda linha, porém, não é capaz de identificar que na terceira linha o último campo está em falta. Reconhece-se que seria possível adaptar a expressão regular para acomodar este caso, porém resultaria numa expressão de fraca interpretabilidade e, provavelmente, suscetível a outras más interpretações, principalmente quando comparada com a simplicidade da utilização da função *split*.

No que concerne dificuldades no desenvolvimento do projeto, salienta-se a inserção de vírgulas de forma a separar os campos num ficheiro no formato json. Tome-se como exemplo o seguinte ficheiro:

```
[
  {
    "nome": "Ana Maria",
    "numeros": [845,5900],
    "notas_avg": 14.54
  },
  {
    "nome": "João Martins",
    "numeros": [850,6000],
    "notas_avg": 14.5
  }
]
```

É necessário ter em conta que *notas_avg* é o último campo dentro de cada bloco, é ainda necessário detetar que o bloco correspondente a João Martins é o último do ficheiro, de forma a não colocar vírgula no final do bloco. Esta tarefa torna-se exponencialmente mais difícil devido à adição de duas funcionalidades: a possibilidade de ter campos em falta e a possibilidade de aplicar várias operações a um só campo.

Para resolver esse problema, é necessária uma flag que indique que o programa se encontra a processar o último campo não vazio de uma linha, flag calculada nas linhas 41 e 46 do anexo C. Para resolver o problema de várias operações no úlimo campo não vazio, essa flag tem ainda de ser combinada com outra condição lógica, como se observa na lina 105 do mesmo anexo.

No que concerne limitações da ferramenta, salienta-se a única detetada, um caso muito específico: a ferramenta é incapaz de processar um ficheiro csv em que a primeira coluna seja um campo de grupo de valores que contenha várias operações. Isto deve-se ao facto de a aplicação, normalmente, inferir o separador dos campos. O problema advém de, quando a primeira coluna contém várias operações serão encontrados ambos os caracteres que podem servir de separação: a vírgula e o ponto e vírgula.

A impossibilidade de detetar qual dos dois é o separador advém do facto de não se conseguir forçar uma expressão regular a devolver sempre o maior match, quando se trata de uma alternativa e ambos os ramos dão match.

Um exemplo básico deste problema será caso tenha a seguinte regex: $(ab|abc)$ e caso tenha a string "abc", o match returnado pelo re.match é "ab". Caso se pretenda que **abc** seja o match retornado tenho de mudar a minha regex para $(abc|ab)$.

Isto é uma solução adequada quando já se sabe previamente o tamanho dos matches na regex, colocando-as por ordem de tamanho de forma a obter o match de maior tamanho, porém, quando o tamanho dos possíveis

matches é indefinido, é impossível saber em que ordem colocar as alternativas na regex de forma a obter o match de maior tamanho.
Um exemplo::

Quero uma regex que seja capaz de detetar se o caracter final de uma secção seja uma vírgula ou um ponto e vírgula, sendo possível ter o outro no meio da string
EX:

aaa;bbb;ccc,ddd → deve retornar o ,
aaa,bbb,ccc;ddd → deve retornar o ;

a regex seria:

```
(?:(?:[^,]+,?)+(;))|(?:(?:[^;]+;?)+(,))
```

Nesta regex, verificar-se-ia se o grupo de captura 1 estava ativo, caso fosse esse o caso, concluiria que o separador final seria ;, se fosse o grupo 2 a dar match, concluiria que o separador final seria ','.
O problema surge numa string em que o separador final é ',' e o carácter separador de operações o ';', pois a expressão dará match na primeira alternativa da regex, em vez da segunda. O match é tecnicamente correto, porém não é o match pretendido, visto não ser o maior match possível.
Não tendo sido encontrada forma de forçar o maior match em regexs com alternativas sobrepostas, fica a inibição da primeira coluna conter vários operadores como uma limitação da nossa ferramenta.

4.5 Testes realizados e Resultados

Mostram-se, de seguida, dois testes realizados, o primeiro centra-se em verificar se o programa consegue sustentar todas as funcionalidades simultâneamente, o segundo pretende medir a eficácia do programa a ler ficheiros csv extensos.
Primeiro teste:

Listing 4.1: Input do primeiro teste

```
numero,nome,number*avg;max;min,abc*,a*group;avg,column_to_cast+
A71823,A,(1;2;3),(124),(99;43.346),12
A89765,BC,,(1;2;3),,213.9
,,,(4576),(99.2;124),-11111111111111.111
-9.000000001,,,,,
```

Listing 4.2: Output do primeiro teste

```
[
        {
                "numero": "A71823",
                "nome": "A",
                "number_avg": 2.0,
                "number_max": 3.0,
                "number_min": 1.0,
                "abc": [124.0],
```

```
9                   "a": [99.0, 43.346],
10                  "a_avg": 71.173,
11                  "column_to_cast": 12.0
12          },
13          {
14                  "numero": "A89765",
15                  "nome": "BC",
16                  "abc": [1.0, 2.0, 3.0],
17                  "column_to_cast": 213.9
18          },
19          {
20                  "abc": [4576.0],
21                  "a": [99.2, 124.0],
22                  "a_avg": 111.6,
23                  "column_to_cast": -11111111111111.111
24          },
25          {
26                  "numero": "-9.000000001"
27          }
28  ]
```

Tempo de execução: **0.001s**

No segundo teste, foi criado um ficheiro csv em que as 4 linhas iniciais do corpo do csv são repetidas até perfazerem 10 000 linhas. Estas linhas são:

Segundo teste:

Listing 4.3: Cabeçalho e quatro primeiras linhas do input do segundo teste

```
1 numero;nome;curso;notas*group,min;campos*avg,max;test+
2 A71823;Ana Maria;MIEI;(12,14,15,18);(15,45);88
3 A89765;Joao Martins;;(11,16.9,13);(1,2,3);99
4 A54321;Paulo Correia;MIEFIS;;;
5 A1234;Jose Esteves;MIEPSI;(20,20,19.5,20);;45
```

Como output é gerado um ficheiro de 80 000 linhas, com sintaxe e conversão correta. Este ficheiro não é apresentado devido à dimensão proibitiva.

Tempo de execução: **6.87s**

Capítulo 5

Conclusão

Conclui-se o presente relatório com uma apreciação positiva do trabalho realizado. A ferramenta desenvolvida é capaz de processar com sucesso diversos ficheiros no formato CSV, suportado um conjunto de funcionalidades adicionais, não sacrificando, para isso, eficácia e velocidade de processamento.

É aplicado, com sucesso, o conhecimento adquirido, até ao momento, na unidade curricular de Processamento de Linguagens [2], com especial ênfase na utilização de expressões regulares para deteção de padrões textuais.

Como futuro trabalho sugere-se o suporte a mais caracteres delimitares de campos e um conjunto mais variado de operações sobre listas de valores, também se pretende a resolução da limitação de processamento de ficheiros CSV em que a primeira coluna seja um campo que contém múltiplas operações, problema relatado na secção 4.4

Apêndice A

Código de processamento do cabeçalho do CSV

Listing A.1: Código de processamento do cabeçalho

```
1 import re
2 from typing import List
3
4 def process_header(header_line: str) -> (str, str, List[str], List[str]):
5         """Retrieves information about the fields declared in the header
6
7         Args:
8                 header_line (str): First line of the csv file
9
10        Raises:
11                NameError: Unsupported operation is found in the header
12
13        Returns:
14                str: field delimiter
15                str: character separating group operations,
16                List(str): List of the names of each column,
17                List(str): List where each index corresponds to the type of operation
                      to be applied to the corresponding column by index
18        """
19
20        column_names = []
21        column_operations = []
22        supported_group_operations = ["group", "sum", "avg", "max", "min"]
23        field_delimiter = re.match(r'?:[^+*;,]+(?:\*|\+)?(?:[^;,]+)?(;|,|(placeholder
              for EOL))',header_line).group(1)
24
25        if field_delimiter == ';':
26                operations_separator = ","
27                captures = re.findall(r'([^+*;]+)(\*|\+)?([^;]+)?', header_line);
28        else:
29                field_delimiter = "," # needs to be set as ',' since (placeholder for
                      EOL) will break the program when processing the body
30                operations_separator = ";"
31                captures = re.findall(r'([^+*,]+)(\*|\+)?([^,]+)?', header_line);
32
```

```
33          for capture in captures:
34                  num_clauses = len(list(filter(None, capture)))
35                  column_names.append(capture[0])
36                  if num_clauses == 1:
37                          column_operations.append("none")
38                  elif num_clauses == 2:
39                          if capture[1] == "*":
40                                  column_operations.append(["group"])
41                          elif capture[1] == "+":
42                                  column_operations.append("cast")
43                  else:  # num_clauses == 3
44                          operations = [operation.lower() for operation in capture[2].
                                split(operations_separator)]
45                          if any([operation not in supported_group_operations for
                                operation in operations]):
46                                  raise NameError("Unsupported Operation in header")
47                          else:
48                                  column_operations.append(operations)
49
50          return field_delimiter, operations_separator,column_names, column_operations
```

Apêndice B

Código de processamento do cabeçalho do CSV (imagem)

```python
def process_header(header_line: str) -> (str, str, List[str], List[str]):
    """Retrieves information about the fields declared in the header

    Args:
        header_line (str): First line of the csv file

    Raises:
        NameError: Unsupported operation is found in the header

    Returns:
        str: field delimiter
        str: character separating group operations,
        List(str): List of the names of each column,
        List(str): List where each index corresponds to the type of operation to be applied to the corresponding
column by index
    """

    column_names = []
    column_operations = []
    supported_group_operations = ["group", "sum", "avg", "max", "min"]
    field_delimiter = re.match(r'?:[^+*;,]+(?:\*|\+)?(?:[^;,]+)?(;|,|$)',header_line).group(1)

    if field_delimiter == ';':
        operations_separator = ","
        captures = re.findall(r'([^+*;]+)(\*|\+)?([^;]+)?', header_line);
    else:
        field_delimiter = "," # needs to be set as ',' since $ will break the program when processing the body
        operations_separator = ";"
        captures = re.findall(r'([^+*,]+)(\*|\+)?([^,]+)?', header_line);

    for capture in captures:
        num_clauses = len(list(filter(None, capture)))
        column_names.append(capture[0])
        if num_clauses == 1:
            column_operations.append("none")
        elif num_clauses == 2:
            if capture[1] == "*":
                column_operations.append(["group"])
            elif capture[1] == "+":
                column_operations.append("cast")
        else:  # num_clauses == 3
            operations = [operation.lower() for operation in capture[2].split(operations_separator)]
            if any([operation not in supported_group_operations for operation in operations]):
                raise NameError("Unsupported Operation in header")
            else:
                column_operations.append(operations)

    return field_delimiter, operations_separator,column_names, column_operations
```

Figura B.1: Função process_header

Apêndice C

Código de processamento do corpo

Listing C.1: Código de processamento do corpo

```
import re
from typing import List

def convert_to_json(csv_lines: List[str],
                                field_delimiter: str,
                                operations_separator: str,
                                column_names: List[str],
                                column_operations: List[str]) -> str:
        """Processes each line of the body of the csv and converts it to a string in
            json format

        Args:
                csv_lines (List[str]): Each line of the csv
                column_names (List[str]): List of the names of each column
                spcolumn_operations (List[str]): List where each index corresponds to
                    the type of operation
                to be applied to the corresponding column by index

        Raises:
                AttributeError: If there's a row with a different number of columns
                    than defined by the header
                AttributeError: If a row contains an empty or missing parenthesis on
                    a group column
                ValueError: If a row contains non-numeric values on a group column
                ValueError: If a row with a cast operation contains non-numeric
                    values

        Returns:
                str: String containing the complete JSON file
        """
        string_list = []

        string_list.append("[")
        for i, line in enumerate(csv_lines):
                string_list.append("\t{")
                fields = line.split(field_delimiter)

```

```
33              if len(fields) != len(column_names):
34                      raise AttributeError(
35                              f"Row {str(i + 2)} does not have the same number of
                                    columns as determined by the header")
36
37              for j, field in enumerate(fields):
38                      if field: # skips empty fields
39                              if column_operations[j] == "none":
40                                      string_list.append(f'\t\t"{column_names[j]}":
                                            "{field}"' +
41                                              ("," if (len(list(filter(None, fields[
                                                    j:])))) > 1) else "")) # condition
                                                    checks if it's not the last non-
                                                    empty field
42                              elif column_operations[j]== "cast":
43                                      try:
44                                              numeric_value = float(field)
45                                              string_list.append(f'\t\t"{
                                                    column_names[j]}": {numeric_value}
                                                    ' +
46                                              ("," if (len(list(filter(None, fields[
                                                    j:])))) > 1) else ""))
47                                      except ValueError:
48                                              raise ValueError(f"Row {str(i + 2)}:
                                                    {field} can't be casted to a
                                                    numeric value")
49                              else:
50                                      values = re.match(r'\((.+?)\)', field) #
                                            extract the values inside the parenthesis,
                                             since it's a list column
51                                      if not values:
52                                              raise AttributeError(f"Row {str(i +
                                                    2)} group column has incorrect
                                                    format")
53
54                                      values = values.group(1).split(
                                            operations_separator)
55
56                                      if len(values) > 0:
57                                              string_list = string_list +
                                                    process_operations(column_names[j
                                                    ], values, column_operations[j], i
                                                    + 2, len(list(filter(None, fields[j
                                                    :])))) == 1) # boolean indicating
                                                    it's the last column of the line
58              if(i == len(csv_lines)-1):
59                      string_list.append("\t}") # the last object doest not have a
                            comma
60              else:
61                      string_list.append("\t},")
62
63      string_list.append("]")
64      return '\n'.join(string_list)
65
```

```
66
67 def process_operations(column_name: str,
68                                         values: List[str],
69                                         operations: List[str],
70                                         row_number: int,
71                                         last_column: bool) -> List[str]:
72
73         """ Converts line portion corresponding to an operations column to it's json
               conterpart
74
75         Args:
76                 column_name (str): Name of the column being processed,
77                 values (List[str]): list of values present in said column,
78                 operations (List[str]): list of operations to be applied to the
                       values in the values list, matched by index,
79                 row_number (int): number of row being processed, useful for throwing
                       informative exceptions
80                 last_column (boolean): flag indicating if it's the last non-empty
                       column \
81                                 of the row being currently processed, for comma
                                       purposes
82
83         Raises:
84                 ValueError:  If there's non-numeric values on a list of values
85
86         Returns:
87                 List[str]: Json portion relative to the operations in the given
                       collumn
88         """
89         operation_results = []
90         try:
91                 numeric_values = [float(value) for value in values]
92                 for i, operation in enumerate(operations):
93                         if operation == "group":
94                                 operation_result = f'\t\t"{column_name}": {
                                       numeric_values}'
95                         if operation == "avg":
96                                 operation_result = f'\t\t"{column_name}_avg": {sum(
                                       numeric_values)/len(numeric_values)}'
97                         elif operation == "sum":
98                                 operation_result = f'\t\t"{column_name}_sum": {sum(
                                       numeric_values)}'
99                         elif operation == "min":
100                                operation_result = f'\t\t"{column_name}_min": {min(
                                       numeric_values)}'
101                        elif operation == "max":
102                                operation_result = f'\t\t"{column_name}_max": {max(
                                       numeric_values)}'
103
104                        operation_results.append(operation_result +
105                         ("" if last_column and i==len(operations)-1 else ",")) #also
                                checks if it's the last operation of the given column
106
107                return operation_results
```

```
108         except ValueError:
109             raise ValueError(f"Non numeric element in row {str(row_number)} in a
                    column that demands such");
```

Apêndice D

Código de processamento do corpo (imagens)

```python
def process_operations(column_name: str,
                       values: List[str],
                       operations: List[str],
                       row_number: int,
                       last_column: bool) -> List[str]:

    """ Converts line portion corresponding to an operations column to it's json conterpart

    Args:
        column_name (str): Name of the column being processed,
        values (List[str]): list of values present in said column,
        operations (List[str]): list of operations to be applied to the values in the values list, matched by index,
        row_number (int): number of row being processed, useful for throwing informative exceptions
        last_column (boolean): flag indicating if it's the last non-empty column \
                               of the row being currently processed, for comma purposes

    Raises:
        ValueError:  If there's non-numeric values on a list of values

    Returns:
        List[str]: Json portion relative to the operations in the given collumn
    """
    operation_results = []
    try:
        numeric_values = [float(value) for value in values]
        for i, operation in enumerate(operations):
            if operation == "group":
                operation_result = f'\t\t"{column_name}": {numeric_values}'
            if operation == "avg":
                operation_result = f'\t\t"{column_name}_avg": {sum(numeric_values)/len(numeric_values)}'
            elif operation == "sum":
                operation_result = f'\t\t"{column_name}_sum": {sum(numeric_values)}'
            elif operation == "min":
                operation_result = f'\t\t"{column_name}_min": {min(numeric_values)}'
            elif operation == "max":
                operation_result = f'\t\t"{column_name}_max": {max(numeric_values)}'

            operation_results.append(operation_result +
             ("" if last_column and i==len(operations)-1 else ",")) #also checks if it's the last operation of the
given column

        return operation_results
    except ValueError:
        raise ValueError(f"Non numeric element in row {str(row_number)} in a column that demands such");
```

Figura D.1: Função process_operations

```python
import re
from typing import List

def convert_to_json(csv_lines: List[str],
                field_delimiter: str,
                operations_separator: str,
                column_names: List[str],
                column_operations: List[str]) -> str:
    """Processes each line of the body of the csv and converts it to a string in json format

    Args:
        csv_lines (List[str]): Each line of the csv
        column_names (List[str]): List of the names of each column
        spcolumn_operations (List[str]): List where each index corresponds to the type of operation
        to be applied to the corresponding column by index

    Raises:
        AttributeError: If there's a row with a different number of columns than defined by the header
        AttributeError: If a row contains an empty or missing parenthesis on a group column
        ValueError: If a row contains non-numeric values on a group column
        ValueError: If a row with a cast operation contains non-numeric values

    Returns:
        str: String containing the complete JSON file
    """
    string_list = []

    string_list.append("[")
    for i, line in enumerate(csv_lines):
        string_list.append("\t{")
        fields = line.split(field_delimiter)

        if len(fields) != len(column_names):
            raise AttributeError(
                f"Row {str(i + 2)} does not have the same number of columns as determined by the header")

        for j, field in enumerate(fields):
            if field: # skips empty fields
                if column_operations[j] == "none":
                    string_list.append(f'\t\t"{column_names[j]}": "{field}"' +
                        ("," if (len(list(filter(None,fields[j:]))) > 1) else "")) # condition checks if it's not
the last non-empty field
                elif column_operations[j]== "cast":
                    try:
                        numeric_value = float(field)
                        string_list.append(f'\t\t"{column_names[j]}": {numeric_value}' +
                        ("," if (len(list(filter(None,fields[j:]))) > 1) else ""))
                    except ValueError:
                        raise ValueError(f"Row {str(i + 2)}: {field} can't be casted to a numeric value")
                else:
                    values = re.match(r'\((.+?)\)', field)  # extract the values inside the parenthesis, since it's
a list column
                    if not values:
                        raise AttributeError(f"Row {str(i + 2)} group column has incorrect format")

                    values = values.group(1).split(operations_separator)

                    if len(values) > 0:
                        string_list = string_list + process_operations(column_names[j], values,
                         column_operations[j],i + 2,len(list(filter(None,fields[j:]))) == 1) # boolean indicating
it's the last column of the line
        if(i == len(csv_lines)-1):
            string_list.append("\t}") # the last object doest not have a comma
        else:
            string_list.append("\t},")

    string_list.append("]")
    return '\n'.join(string_list)
```

Figura D.2: Função convert_to_json

Apêndice E

Código de leitura e processamento de input do utilizador

Listing E.1: Código de leitura e processamento de input do utilizador

```
import sys


if len(sys.argv) == 1:
        input_file_path = "input/data.csv"
        output_file_path = f"output/data.json"
elif len(sys.argv) == 2:
        input_file_path = f"input/{sys.argv[1]}"
        output_file_path = f"output/data.json"
elif len(sys.argv) == 3:
        input_file_path = f"input/{sys.argv[1]}"
        output_file_path = f"output/{sys.argv[2]}"
else:
        raise ValueError("Wrong number of arguments.\nUsage: python main.py [
            input_file_name] [output_file_name]")


# Reading the file
file = open(input_file_path)
lines = file.read().splitlines()  # splitlines to remove \n
file.close()

if len(lines) < 2:
        raise Exception("Insufficient lines in csv")

# Processing the file
start = time.time()
field_delimiter, operations_separator, csv_column_names, csv_column_operations =
    process_header(lines[0])
json_txt = convert_to_json(lines[1:], field_delimiter, operations_separator,
    csv_column_names, csv_column_operations)
end = time.time()

# Writing to json
output_file = open(output_file_path, "w")
```

```
33 output_file.write(json_txt)
34 output_file.close()
35
36 print(f"Time spent processing the csv: {end - start}s")
```

Bibliografia

[1] Re library documentation. https://docs.python.org/3/library/re.html. Consultado: 2021-03-21.

[2] Pedro Rangel Henriques. Documentação da unidade curricular de pl. https://docs.google.com/document/d/1rvaZ2400C60EnWTJo3L7Lex8ONTxFW5Nh7WcGgiyORw/. Consultado: 2021-03-21.