Importação, Normalização e Segmentação de Dados

0. Contexto do tema do Projeto

A Federação Portuguesa de Basquetebol (http://www.fpb) mantem um conjunto de informação detalhada sobre cada jogo (pontos, faltas, assistências e defesas etc..). Esta informação pode ser consultada em:
http://www.fpb.pt/fpb2014/!site.go?s=1&show=my.estatisticas&codigo=Estatisticas.

Os dados são recolhidos em cada jogo, e posteriormente tratados estatisticamente, sendo possível obter indicadores de desempenho por jogo e por jogador.

1. Objetivo do Projeto

À semelhança do Mini-Projeto 1, pretende-se desenvolver um programa em C para extrair informação útil de ficheiros com dados sobre os jogadores e jogos de basquetebol, através de um interpretador de comandos.

Uma das funcionalidades mais pertinentes visa encontrar perfis de jogadores com as mesmas caracteristicas, usando o algoritmo de *clustering* K-Means que gera automaticamente conjuntos de jogadores. Em cada conjunto (*cluster)* vão encontrar-se jogadores com características muito semelhantes.

1.1 Representação de um jogador em Memória

Cada jogador é representado, obrigatoriamente, pela estrutura de dados apresentada na Figura 1, sendo *Date* um tipo de dados apropriado para guardar uma data.

```
typedef struct date {
   unsigned day, month, year;
} Date;



typedef struct statistics {
   float twoPoints; /* total ou media de cestos de dois pontos */
   float threePoints; /* total ou media de cestos de três pontos */
   float assists; /* total ou media de assitencias */
   float fouls; /* total ou media de faltas */
   float blocks; /* total ou media de blocos */
   int gamesPlayed; /* total de jogos disputados */
    /* pode acrescentar algum atributo/campo se achar relevante */
} Statistics;
```

```
typedef struct player {
   int id;
   char name[50];
   char team[50];
   Date birthDate;
   char gender;
   Statistics statistics; /* valores acumulados para todos os jogos disputados */
} Player;
```

Figura 1 – Tipo de dados Player.

Em relação ao Mini-Projeto 1, os dados serão guardados de forma unificada, i.e., cada jogador possuirá, na sua informação, os seus dados estatísticos para a época inteira.

Note que, por exempo, *twoPoints* é do tipo de dados *float*. Isto permitirá guardar quantidades inteiras e reais, consoante o necessário (depende da funcionalidade pretendida).

Na implementação dos comandos descritos neste enunciado podem definir/utilizar outros tipos de dados auxiliares que achem úteis para a resolução dos problemas.

1.2 Dados de entrada

O formato dos ficheiros de entrada é o mesmo que no Mini-Projeto 1. Existem dois tipos de ficheiro com dados:

- Ficheiro de dados sobre os Jogadores;
- Ficheiros de dados sobre os desempenhos dos jogadores nos jogos.

Ambos os ficheiros se encontram em formato CVS.

Ficheiro dos jogadores (cada linha corresponde a informação sobre um jogador

```
<id_player>; <name>; <team>; <birth_date>; <gender>;
...
```

Ficheiro do desempenho dos jogadores em jogos (cada linha corresponde a informação sobre o desempenho de um jogador num jogo)

```
<id_player>; <id_game>; <two_points>;<three_points>;<assists>;<fouls>;<blocks>;
...
```

O valor <*birth_date*> encontra-se no formato "dd/mm/aaaa".

Pode-se assumir que não existem ficheiros "mal-formados".

Neste projeto os ficheiros serão lidos aos pares e a informação composta desses ficheiros deve ser guardada nos tipos de dados definidas em 1.1.

1.2.1 Ficheiros Disponibilizados

Juntamente com este enunciado são disponibilizados 2 ficheiros de entrada para testes:

- players.csv
- games.csv

Após a divulgação do enunciado será disponibilizado no Moodle um exemplo dos resultados esperados na execução da aplicação para estes ficheiros.

1.3 Utilização de TADs

É obrigatória a manutenção em memória da informação importada exclusivamente numa instância do TAD List, sendo ListElem o tipo de dados definido em 1.1.

Cada uma das operações detalhadas na secção seguinte poderá indicar a obrigatoriedade de criação de novas instâncias de TAD List e/ou TAD Map.

A não aderência a estas obrigatoriedades invalida a cotação total das implementações correspondentes.

Não é permitido alterar as interfaces lecionadas dos TAD, nomeadamente os ficheiros list.h e map.h!

1.4 Comandos

Cada comando é representado por uma palavra que pode ser escrita pelo utilizador em maiúsculas ou em minúsculas. **Para todos os comandos, exceto LOAD, caso não existam dados carregados em memória (não lidos ou apagados), o comando deve acusar na consola "SEM DADOS CARREGADOS"**.

Os comandos são os seguintes:

- **LOAD**

  Importa a informação de um par de ficheiros cujo os nomes são solicitados ao utilizador (e.g., players.csv e games.csv). Ambos têm de existir; caso algum não exista a operação não é executada.

  Após o comando, deverá ser apresentado na consola uma mensagem, e.g., **"Foram lidos <N> jogadores e informação sobre <M> jogos"**

  A importação dos dados deve ter em conta o especificado em 1.1 e 1.3, sendo acumulados na estrutura *Player* (no atributo *statistics*) todos os valores de desempenho correspondentes a esse jogador em todos os jogos que ele realizou. O *gamesPlayed* deve ser calculado também.

  Após a importação, os dados lidos nunca devem ser modificados, utilizará cópias dos dados para qualquer manipulação necessária (isto é descrito nos próximos comandos).

- **CLEAR**

  Limpa a informação atualmente em memória. Deverá indicar o número de registos que foram descartados, e.g., **"Foram apagados <N> registos"**

- **SHOW**

  Mostra a informação dos jogadores no formato em que ela é guardada em memória.

- **QUIT**

  Termina a aplicação e liberta qualquer memória ainda alocada.

- **SORT**

Mostra os jogadores importados de forma ordenada crescentemente. O critério de ordenação deve ser solicitado ao utilizador e pode ser um dos seguintes:

- Nome
- Data de nascimento (desempate por nome);
- Número de jogos jogados (desempate por nome);

Na implementação do comando, todos os dados deverão ser copiados para uma instância auxiliar do TAD List e a ordenação deverá ser efetuada sobre esta instância; os resultados deverão ser mostrados com a operação *listPrint* do TAD List.

- **AVG**

Calcula médias das estatísticas de cada jogador importado, nomeadamente média de cestos por jogo (de 2 e de 3 pontos), faltas, assistências e blocos;

A informação dos jogadores e respetivos valores médios deverá ser calculada e guardada temporariamente num TAD List auxiliar através de uma função auxiliar

```
PtList averageStatistics(PtList players)
```

Esta função será reutilizada para o comando TYPE (mais à frente).

O comando, posteriormente, mostra a listagem dos jogadores ordenada decrescentemente pelo índice MVP médio (*avgMVP*) de cada jogador:

avgMVP = 3 x <avg treePoints> + 2 x <avg twoPoints> + <avg assists> + 2 x <avg blocks> – 3 x <avg fouls>;

- **NORM**

Mostra todos os jogadores e as suas estatísticas normalizadas entre 0 e 1, segundo a normalização *min-max*.

Cada valor é calculado por $x' = \frac{x - min_i}{max_i - min_i}$,sendo min_ie max_i os valores do mínimo e máximo de cada estatística *i* entre todos os jogadores, i.e., os valores máximo/mínimo de *twoPoints* entre todos os jogadores permite normalizar os valores individuais de cada um no intervalo pretendido.

A implementação deste comando deverá utilizar uma função auxiliar

```
PtList normalizeStatistics(PtList players)
```

que recebe a lista de jogadores importados e devolve uma nova instância com cópias dos jogadores, mas cujas estatísticas estão normalizadas no intervalo [0,1].

Esta função deverá ser reutilizada no comando KMEANS (mais à frente).

- **TYPE**

Nota preliminar: Esta função deve operar com valores médios por jogador, i.e., os obtidos através da função *averageStatistics* do comando AVG. Isto por forma à segmentação ser independente do número de jogos disputados pelos jogadores.

Segmenta os jogadores em 3 tipos (sub-conjuntos) de acordo com as seguintes características:

- Tipo ***Shooting-Guard***: Todos os jogadores cujos valores médios *twoPoints* e *threePoints* sejam superiores à média geral, e cujos valores médios *assists* e *blocks* sejam inferiores à sua respetiva média geral.

- Tipo ***Point-Guard***: Todos os jogadores cujos valores médios *twoPoints* e *threePoints* sejam inferiores à média geral, e cujos valores médios *assists* e *blocks* sejam superiores à média geral.

- Tipo ***All-Star***: Todos os jogadores cujos valores médios *twoPoints*, *threePoints*, *assists* e *blocks* sejam superiores à média geral.

Em termos de implementação, cada sub-conjunto de jogadores deverá estar contido numa instância auxiliar de TAD List, cujos conteúdos são depois mostrados, i.e., instâncias separadas possuiem cópias de todos os jogadores de um dos tipos (*All-Star, Point-Guard e Shooting-Guard*).

- **CHECKTYPE**

Segmenta os jogadores da mesma forma que no comando "TYPE", mas depois utiliza um TAD Map para mapear o id de um jogador para informação sobre o seu tipo. O "valor" mapeado deve ser de um tipo tal (definido pelo(s) aluno(s)) que permita guardar:

- type (cadeia de carateres: {“shooting-guard”, “point-guard” ou “all-star”})
- avgTwoPoints; /* media do jogador */
- avgAllPlayersTwoPoints; /* média geral */
- … (atributos respetivos para threePoints)
- avgBlocks,
- avgAllPlayersBlocks
- ... (atributos respetivos para assists e fouls)

isto é, contém o tipo de jogador, valores individuais e médias globais;

O programa deve então proceder à solicitação de vários ids de jogador (chave do mapa) e mostrar o valor associado; isto até que seja introduzido um valor id negativo (id inválido).

A pesquisa da informação associada deve ser feita exclusivamente através das operações do TAD Map.

- **KMEANS**

Segmenta os jogadores em K *clusters* através do algoritmo K-Means (ver Anexo I).

Nota: Este algoritmo deve operar com dados dos jogadores já normalizados, obtidos a partir da função auxiliar *normalizeStatistics* do comando NORM.

Deve ser solicitado ao utilizador:

- número de clusters desejados (valor *K*);
- máximo de iterações do algoritmo a efetuar (valor *maxIterations*);
- variação mínima do erro entre iterações (valor *deltaError*);

Cada *cluster* deve ser representado obrigatoriamente pelo seguinte tipo de dados:

```
typedef struct cluster {
   float meanTwoPoints;
   float meanThreePoints;
   float meanAssists;
   float meanFouls;
   float meanBlocks;
   PtList members;
} Cluster;
```

Sendo os 5 primeiros atributos os valores do *centróide* do *cluster* e o atributo *members* uma lista de jogadares que pertencem ao cluster.

Na implementação do algoritmo deverá utilizar um *array* do tipo *Cluster* (alocado dinamicamente, consoante o número de clusters K pretendidos) para manter todos os clusters em memória;

Existem dois critérios de paragem: número de iterações e variação do erro entre iterações. Só terá cotação total se implementar ambos, embora o algoritmo possa funcionar fazendo apenas um determinado número de iterações.

Nota: para efeitos de teste, sugerem-se os seguintes valores:

- K = {3, 5, 6}, *maxIterations* = 20, e; deltaError = 0.001.

Após o clustering/segmentação dos jogadores, o comando deve apresentar os dados de cada cluster.

2 Relatório

No relatório deverão constar as seguintes secções (para além de capa com identificação dos alunos e índice):

a) Descrição dos TAD utilizados, descrição da implementação e justificação da estrutura de dados escolhida para a implementação;
b) Para cada comando (exceto CLEAR,SHOW e QUIT) fornecer:
   - O código/função respeitante à funcionalidade;
   - A complexidade algorítmica da respetiva implementação;
c) Limitações: Quais os comandos que apresentam problemas ou não foram implementados;
d) Conclusões: Análise crítica do trabalho desenvolvido.

3 Tabela de Cotações e Penalizações

A avaliação do trabalho será feita de acordo com os seguintes princípios:

- Estruturação: o programa deve estar estruturado de uma forma modular e procedimental;
- Correção: o programa deve executar as funcionalidades, tal como pedido.
- Legibilidade e documentação: o código deve ser escrito, formatado e comentado de acordo com o standard de programação definido para a disciplina.

A nota final obtida, cuja tabela de cotações se apresenta a seguir, será ponderada de acordo com os princípios acima descritos.

Funcionalidade/Comando	**Cotação**
LOAD + SHOW + CLEAR + QUIT	4
SORT	2
AVG	1,5
NORM	1,5
TYPE	2
CHECKTYPE	2
KMEANS	
Utilização de número de iterações como único critério de paragem	3
Utilização simultânea do número de iterações e erro mínimo	4
Relatório	3
TOTAL	20 valores

A seguinte tabela contém penalizações a aplicar:

Descrição	**Penalização**
Uso de variáveis globais	até 2
Não separação de funcionalidades em funções/módulos	até 3
Não libertação de memória	até 3
Não utilização dos TAD obrigatórios	anulado

4 Instruções e Regras Finais

O IDE a utilizar fica ao critério dos alunos, mas, caso não utilizem o IDE usado na disciplina (i.e., Visual Studio), terão que, **antes de submeter, criar os respetivos projetos finais no IDE** Visual Studio 2017.

O não cumprimento das regras a seguir descritas implica uma penalização na nota do trabalho prático. Se ocorrer alguma situação não prevista nas regras a seguir expostas, essa ocorrência deverá ser comunicada ao respetivo docente de laboratório de ATAD.

Regras:

a) O Mini-Projeto deverá ser elaborado por **dois alunos do mesmo docente de laboratório**.

b) A nota do Mini-Projeto será atribuída individualmente a cada um dos elementos do grupo após a discussão. As discussões poderão ser orais e/ou com perguntas escritas. As orais poderão ser feitas com todos os elementos do grupo presentes em simultâneo ou individualmente.

c) A apresentação de relatórios ou implementações plagiadas leva à imediata atribuição de nota zero a todos os trabalhos com semelhanças, quer tenham sido o original ou a cópia.

d) No rosto do relatório e nos ficheiros de implementação deverá constar o número, nome e turma dos autores e o nome do docente a que se destina.

e) O trabalho deverá ser submetido no moodle, no link do respetivo docente de laboratórios criado para o efeito, até às **11:00 do dia 11 de Junho**. Para tal terão que criar uma pasta com o nome: **nomeAluno1_númeroAluno1-nomeAluno2_númeroAluno2**, onde colocarão o ficheiro do relatório em formato **pdf** e uma pasta com o projeto Visual Studio da implementação das aplicações a desenvolver. Os alunos terão de submeter essa **pasta compactada em formato ZIP**. Apenas será permitido submeter um ficheiro.

f) Não serão aceites trabalhos entregues que não cumpram na íntegra o ponto anterior.

g) As datas das discussões serão publicadas após a entrega dos trabalhos.

Anexo I

K-Means – Clustering

O algoritmo *k*-Means é um método de análise de clusters usado em *data mining*. O seu objetivo é particionar N observações em K conjuntos (clusters) de forma que cada observação pertença à média de observações mais próxima (um cluster). Cada observação possui p características.

Em cada iteração do algoritmo K centróides são recalculados. Cada centróide corresponde à média das observações existentes num cluster, algo similar ao centro de massa.

Tendo em conta que as características de cada observação podem ter várias escalas, elas devem ser normalizadas de forma a não desvirtuar o conceito de distância euclediana.

No caso do projeto, cada observação corresponde a um jogador com os dados normalizados e cada característica é uma das suas estatísticas (excluindo numberGames). O propósito é utilizar este algoritmo para segmentar os jogadores existentes por grupos com características similares.

O k-Means tem como entrada principal N observações, e o seu algoritmo é descrito pelo seguinte pseudo-código:

```
Algoritmo Kmeans
   Input:  lista – List de observações com os valores normalizados
           K - número de clusters, inteiro > 2
           maxIterations – número máximo de iterações a realizar, inteiro > 1
           deltaError – variação mínima do erro entre iterações, valor real

   Output: array de clusters

BEGIN

    Inicializa os K clusters, atribuindo a cada cluster o valor do centróide igual ás
    características de uma observação escolhida aleatoriamente.

    iterationNumber <- 1
    prevError <- INFINITO POSITIVO (ou valor real muito grande)
    iterationError  <- INFINITO POSITIVO (ou valor real muito grande)

    DO
        IF iterationNumber > 1 THEN
           prevError <- iterationError
        END IF

        Atribui cada observação ao cluster que fica "mais perto" (usar a distância
        euclidiana).

        Recalcula cada um dos K centróides. Este cálculo é feito, para cada centróide,
        como sendo a média para todos as observações pertencentes ao respetivo cluster,
        de cada uma das suas características.

        Calcula o erro para a iteração atual - iterationError
        Para calcular este erro calcula-se o quadrado da distância de cada observação
        em relação ao centróide do cluster a que pertence, e depois soma-se para todos
        as observações.

        iterationNumber <- iterationNumber + 1

    WHILE (iterationNumber < maxIterations) AND abs(prevError – iterationError) >
deltaError)

    RETURN array com k clusters
END
```

A distância Euclidiana entre duas observações **a** e **b**, com **p** características, define-se por:

$$d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_p - b_p)^2}$$

O erro (E) de uma iteração do algoritmo é calculado por:

$$E = \sum_{j=1}^{c} \sum_{x_i \in w_j} d(x_i, m_j)^2$$

onde:

$\boldsymbol{x_i}$ é a observação na posição i
$\boldsymbol{w_j}$ é o cluster j
$\boldsymbol{m_j}$ é o centróide do cluster j
$\boldsymbol{d(x_i, m_j)}$ é a distância eucledian entre a observação $\boldsymbol{x_i}$ e o centróide $\boldsymbol{m_j}$.

Nos recursos:

- https://en.wikipedia.org/wiki/K-means_clustering;
- http://stanford.edu/class/ee103/visualizations/kmeans/kmeans.html

podem visualizar-se demonstrações gráficas do funcionamento do algoritmo; elas são facilmente visíveis quando o número de características p = 2.

(fim de enunciado)