Primeira Fase do Trabalho de Recuperação de Informação — v.15/10/2020

Caros alunos, nesta primeira fase do trabalho de Recuperação de Informação vocês devem montar um primeiro Sistema de Recuperação de Informação que venha a indexar um conjunto de dados bem desestruturados: os arquivos textos que vocês possuem. Vocês deverão elaborar dois programas: o *indexador do Meu sistema de Recuperação de Informação*, `mir`; e o *Buscador do Meu sistema de Recuperação de Informação*, `mirs`.
Deve-se escrever os utilitários em python 3 e usar o pacote `argparse` para processar as opções de linha de comando.

***

O indexador `mir` recebe na linha de comando o nome de um `diretório` cuja sub árvore de diretório nele pendurada será indexada. Primeiramente o utilitário deve listar e enumerar recursivamente todos os arquivos com a extensão `.txt` que se encontram na sub árvore do `diretório` em questão. Por exemplo, ao executar a linha de comando `./mir.py diretório` depois de descompactar a minicoleção de documentos fornecida, o indexador deve produzir uma saída como a seguinte:

```
Lista de arquivos .txt encontrados na sub-árvore do diretório: diretório
0 arq0.txt
1 arq1.txt
2 arq2.txt
3 subdirA/arq3.txt
4 subdirA/arq4.txt
5 subdirA/subdirB/arq5.txt
6 subdirC/arq6.txt
7 xarq7.txt
8 zoutroarq8.txt
Foram encontrados 9 documentos.
```

Uma *lista de arquivos* armazenará seus nomes, que serão enumerados e identificados pelos respectivos índices na lista. Esta lista efetivamente mapeia estes identificadores nos caminhos dos arquivos da subárvore, relativos ao diretório-raiz fornecido ao indexador. (Por exemplo, o identificador `6` é mapeado na cadeia `subdirC/arq6.txt`, o caminho do respectivo arquivo já relativo a `diretório`.) Os números desta enumeração fazem o papel de *identificadores* únicos destes arquivos. Observe que os caminhos dos arquivos presentes na lista de arquivos não são nomes de arquivos aptos a serem abertos porque são relativos ao diretório-raiz `diretório`. Assim é necessário juntar (vide `os.path.join`) o diretório-raiz e o caminho relativo para se obter um nome de arquivo apto a ser aberto. (No exemplo considerado, `diretório/subdirC/arq6.txt`.)
Para produzir a lista de arquivos acima foram usados métodos do pacote `os` como `os.listdir`, `os.path.isfile`, `os.path.isdir`, `os.path.join`, `os.path.splitext`, de modo que se recomenda olhar a documentação destes métodos e do pacote. Os métodos `os.path.islink`

e `os.path.realpath` também são úteis para tratar links simbólicos para arquivos e diretórios. Também recomendamos que se observe a documentação de `os.path.relpath`, `os.path.isabs` `os.path.getsize`, `os.path.getmtime`, e `os.stat`, seja para implementação desta fase, seja em vista de fases futuras.

Em seguida, o utilitário deve indexar os documentos e criar o índice invertido, que a cada termo associa o conjunto dos documentos que o contém. Adotar-se-á a solução usual de um arquivo invertido, como proposto no livro, construindo um dicionário a conter o vocabulário dos documentos, e associando a cada termo (token) a lista de incidência com os identificadores únicos dos documentos que o contêm. No exemplo oferecido, disponível na página da disciplina, oferecemos um conjunto de nove arquivos oriundos de páginas da Agência Fapesp de Notícias, com artigos antigos e novos. A indexação deste exemplo deve associar: ao token `ime`, obtido do termo `IME`, a lista com todos os oito arquivos não vazios, `[0, 1, 2, 3, 4, 6, 7, 8]`; ao termo `computação`, a lista `[1, 2, 3, 4, 6, 7, 8]`; e ao termo `prêmio`, a lista de incidência `[1, 2, 4, 6, 7]`.

Os arquivos-texto podem estar escritos em diferentes línguas e estender a tabela ASCII de diversas formas (UTF-8, UTF-16, ISO-8859-1, Windows-1252, etc), mas internamente os tokens devem ser todos codificados em utf-8 e serão *convertidos para letras minúsculas*. O arquivo `arq0.txt` deveria conter as palavras `prêmio` e `computação`, mas problemas de decodificação bagunçaram todas as palavras com letras acentuadas e o identificador associado `0` não aparece nas listas de incidência destes termos. É útil, pois, que ao reconhecer uma opção `-v` de linha de comando, o indexador relate informações mais verborrágicas que descrevam para cada arquivo: o seu identificador único; o encoding usado na decodificação do arquivo; a confiança na deteção deste enconding; o tamanho do arquivo; e o seu caminho relativo ao diretório raiz da indexação. O indexador `mir` deve implementar esta opção `-v`.

Nesta fase do projeto, *não é necessário armazenar a frequência dos termos nos documentos nem montar um índice posicional*, recomendado no caso de busca por frase ou busca por proximidade. Depois de montado o dicionário, deve ser escrito um sumário com informações como as seguintes:

```
Os 9 documentos foram processados e produziram um total de 9537 tokens,
que usaram um vocabulário com 1304 tokens distintos.
Informações salvas em diretório/mir.pickle para carga via pickle.
```

Devem ser armazenadas via `pickle`, nesta ordem:

1. a *cadeia* de caracteres 'MIR 1.0';
2. a *lista de arquivos* com os caminhos dos documentos, relativos ao `diretório` fornecido;
3. o *dicionário* que mapeia os tokens do *vocabulário* em suas respectivas listas de incidência, as listas ordenadas com os identificadores dos arquivos que contêm os tokens;
4. e um *dicionário* que mapeia cada arquivo da lista no seu esquema de codificação de texto, o seu *encoding*.

Note que tanto o índice invertido quanto o sumário acima dependem não apenas da lista de arquivos – em nosso exemplo, `['arq0.txt', 'arq1.txt', 'arq2.txt', 'subdirA/arq3.txt', 'subdirA/arq4.txt', 'subdirA/subdirB/arq5.txt', 'subdirA/subdirB/subdirE/arq6.txt', 'xarq7.txt', 'zoutroarq8.txt']` – e do respectivo esquema de codificação usado para decodificar cada um deles como também do processo de tokenização dos arquivos já decodificados, assuntos aos quais dedicamos a seção seguinte.

***

Com relação à codificação de arquivos texto, tem se tornado padrão o uso da extensão da tabela ASCII conhecida como Unicode, bem como o uso de sua codificação mais popular, o UTF-8. O Unicode unifica num único código os alfabetos de todas as línguas usadas no mundo, superando hoje mais de cento e quarenta mil símbolos, de modo que seu uso requer que se quebre a convenção de que cada caractere é representado por um único byte. Em python 3, quando se abre para leitura um arquivo em modo texto com `open( NomeDoArquivo, 'r' )`, é implícita a codificação padrão `encoding='utf-8'`.
Quando fomos preparar este trabalho e codificar uma solução para ele, deparamo-nos com uma dificuldade: em muitos arquivos, a leitura do arquivo texto produzia palavras que acusavam erros do tipo `UnicodeDecodeError` em muitas palavras. Ocorre que dos milhares de arquivos `.txt` que temos acumulando há algumas décadas, apenas os 7% que usam o formato ASCII e os 49% que usam o formato UTF-8 foram decodificados. Nos 44% que usam o formato ISO-8859-1 e nos 0.2% que usam sua extensão Windows-1252, uma exceção com o erro acima era levantada e o indexador abortava. Mesmo quem tenha uma história computacional não tão longa pode vir a enfrentar o mesmo problema caso tenha baixado algum arquivo na internet produzido a partir de material mais antigo. Esta é uma dificuldade inerente à manipulação de textos escritos em diversas línguas e diversas codificações. Parte da solução que propusemos faz uso do pacote `chardet`:

```
import chardet
def GetFileEncoding(file_path):
    """ Get the encoding of file_path using chardet package  """
    with open(file_path, 'rb') as f:
        return chardet.detect( f.read() )
```

E a leitura de texto pode ser feita usando o *encoding* detectado, como também foi sugerido:

```
...
enc = GetFileEncoding( arquivo )
encoding = enc['encoding']
confidence = float(enc['confidence'])*100
print( '%5d %-10s %4.1f%% %s' % (identificador, encoding, confidence, arquivo) )
if confidence < 63: myerr = 'replace'
else: myerr = 'strict'
with open( arquivo, 'r', encoding=encoding, errors=myerr ) as filehandle:
```

```
    # Lê arquivo texto, quebrando em linhas e em palavras, gera tokens,
    # insere-os no dicionário e atualiza as listas de incidência
    ...
```

Infelizmente, a detecção acima ainda não garante a corretude da decodificação em todas as situações. Arquivos texto grandes, 'majoritariamente' `'utf-8'`, mas que sofreram a inserção de caracteres inválidos de codificação `'Windows-1252'`, são facilmente reconhecidos como sendo desta codificação ou de variantes assemelhadas. Isto também diminui o índice de confiança atribuído à codificação detectada pelo pacote `detect` e, por esta razão, o código proposto relaxa o rigor no tratamento de erros de decodificação de modo a substituir caracteres inválidos pelo símbolo `'?'`, o que é feito no código acima pela substituição de `'strict'` por `'replace'` quando a confiança for abaixo do limiar de `63%`. Verdade seja dita, isto não corrige satisfatoriamente o problema, mas evita a geração de um sinal de exceção e o consequente abortamento do programa ...
Além de quebrar a linha em palavras eliminando-se os espaços, é necessário eliminar os caracteres de pontuação que tipicamente são "colados" nas extremidades de uma palavra. Contudo, observamos muitos caracteres estranhos no interior das palavras formadas, de modo que a um certo momento sugerimos esta tokenização:

```
def StripPonctuation(s):
    # return #s.translate(mytable)
    tokens = re.sub(r'([^\w\s]|\d|_)+', ' ', s)
    return tokens.split()
```

Este código usa o pacote de expressões regulares da Python e eliminaria da linha qualquer caractere estranho à constituição de uma palavra, tanto um caractere da tabela ASCII quanto do alfabeto estendido.[1] Também foi sugerida a exploração de outras soluções que usassem `re.finditer`, `re.split`, `str.translate`, etc., o que está feito a seguir, onde a tokenização é feita pelo método `split` definido no módulo de expressões regulares:

```
  import re
  resplit= re.compile( r'[\W\d_\s]+' )
  ...
  file_rel_path = lista_de_nomes_relativos_dos_arquivos[ fileID ]
  enc = GetFileEncoding( ref_dir, file_rel_path )
  arquivo = os.path.join( ref_dir, file_rel_path )
  if verborragico:
      print( '%5d %-10s %4.1f%% %6d  %s' % (fileID, enc['encoding'],
             float(enc['confidence'])*100, os.stat( arquivo ).st_size,
             file_rel_path) )
```

[1] A função substitui (`re.sub`) por espaço (`' '`) qualquer sequência não vazia (`'+'`) de caracteres que: não (`^`) fazem parte de identificadores (`\w`) nem sejam tratados como espaços (`\s`); nem tampouco (`|`) que sejam dígitos (`\d`) ou (`|`) sublinhado (`_`). As letras do alfabeto, acentuadas ou não, os dígitos e o sublinhado são caracteres aceitos como parte dos identificadores em python.

```
with open( arquivo, 'r', encoding = enc['encoding'],
           errors = enc['errors'] ) as file_handle:
    # Quebrando texto em linhas e em palavras separadas por
    # cadeias que casam com resplit
    words_gen = ( word.lower() for line in file_handle
                               for word in resplit.split(line) )
    for token in words_gen: # words_gen is an iterator
        # Manage token ...
```

Ao invés de quebrar uma linha em palavras por conta dos espaços para depois eliminar a pontuação remanescente, o código já quebra em palavras separadas por sequências de um ou mais (`+`) caracteres (`[ ... ]`): que *não* sejam usadas em identificadores (`\W`); ou que sejam dígitos (`\d`); ou sublinhado (`_`); ou que sejam tratados como espaços (`\s`). Ainda que não seja perfeita, *a tokenização neste trabalho deve ser feita desta forma*, também por razões de padronização.

Mesmo que a tokenização acima pareça eliminar todos os caracteres que não sejam tratados como integrantes de uma palavra, a verdade é que ela não elimina o problema de uma codificação de texto mal detectado que faz com que a palavra `prêmio` e `matemática` sejam decodificadas como `prÃªmio` e `matemÃ¡tica`, respectivamente, como acontece no arquivo `arq0.txt` fornecido. Ademais, a tokenização adotada faz com que o sinal de pontuação `¡` divida a palavra `matemÃ¡tica` em duas: `matemÃ` e `tica` ...

Observe que na implementação deste trabalho não estão sendo inseridos nem números nem datas no vocabulário e, como prometido, a corrente atualização do enunciado do trabalho fornece um arquivo `.tgz` de uma árvore de diretório com alguns poucos arquivos, tirados de exemplo reais. Estes exemplos, porém, não reproduzem a dificuldade de uma situação onde se possui milhares de arquivos-texto, dos quais apenas algumas dezenas possuem uma codificação mal reconhecida a ponto de poluir o vocabulário com mais de cinquenta, mil palavras à primeira vista inválidas por usarem "símbolos estranhos" ao alfabeto usado nas Línguas em questão.

Como afinal se pode processar tanta informação desestruturada sem nem ao menos poder rodar `SELECT` num banco de dados `SQL`?

Numa coleção de documentos real – como a que temos para alimentar este sistema de recuperação de informação – o cientista da computação deve resolver o problema como este de procurar num vocabulário de cerca de oitocentas mil palavras quais são as cinquenta mil que apresentam caracteres inesperados na Língua e quais são as dezenas de arquivos, perdidos em meio a milhares, que estão sendo decodificados incorretamente e poluindo o vocabulário. As especificações deste projeto também buscam levar em consideração estas dificuldades e facilitar o trabalho de preparação dos dados desestruturados extraídos da coleção de documentos.

Em particular, deve-se também implementar a opção `-t topo`, no buscador `mirs`, de forma a listar os `topo` tokens mais frequentes nos documentos, com maior comprimento da lista de incidência (índice DF). Use a ordem alfabética como segunda chave de ordenação. Sugere-se considerar o uso das funções `itemgetter` e `attrgetter` do pacote `operator`. Para o exemplo

fornecido, nossa implementação oferece à linha de comando `mirs.py -t 3 diretório/` a seguinte saída:

```
MIR (My Information Retrieval System) de diretório/mir.pickle com 1304 termos e 9 docume

  DF  Termo/Token     Lista de incidência com IDs dos arquivos
   8  a               [0, 1, 2, 3, 4, 6, 7, 8]
   8  agencia         [0, 1, 2, 3, 4, 6, 7, 8]
   8  as              [0, 1, 2, 3, 4, 6, 7, 8]
Listados 3 tokens, ordenados decrescentemente por freq. de documento (DF).
```

Ademais, a listagem com os `topo` melhores admite a especificação de dois tipos de filtros para os tokens listados: aqueles que satisfazem a expressão regular `regex` oferecida pela opção `-r regex`; e os tokens que NÃO satisfazem a expressão regular `regexneg` especificada pela opção `-R regexneg`. Por exemplo, a execução seguinte lista nos dados do exemplo os três tokens mais frequentes que NÃO são codificados pelas letras do alfabeto latino codificadas na tabela ASCII (`[a-z]`) unidas às letras acentuadas `[À-ÿ]` presentes no padrão ISO-8859-1 e enumeradas de `'\u00c0'` a `'\u00ff'`. O uso dos caracteres especiais circunflexo e dólar requer que toda a cadeia do token pertença à linguagem associada à expressão regular.

```
[user@local:~/] mirs.py diretório -t 3 -R '^[a-zÀ-ÿ]+$'
MIR (My Information Retrieval System) de diretório/mir.pickle com 1304 termos e 9 docume

Palavras que NÃO satisfazem a REGEX "^[a-zÀ-ÿ]+$"
Total:   1304 Não regex:     25 Regex:   1279 Razão: 51.160

   DF  Termo/Token     Lista de incidência com IDs dos arquivos
    4  º               [1, 2, 7, 8]
    3  ª               [6, 7, 8]
    1  agãªncia        [0]
Foram listados 3 tokens de 6 arquivos NÃO satisfazendo regex "^[a-zÀ-ÿ]+$".
[user@local:~/]
```

A execução do programa relata 25 tokens com caracteres estranhos, alguns dos quais com dificuldade de serem aceitos no ambiente verbatim do LaTeX. Além dos tokens `ª` e `º`, presentes em cinco arquivos, os demais 23 ocorrem apenas no arquivo `arq0.txt`.
Além das opções acima descritas, o buscador `mirs` recebe como parâmetros na linha de comando o `diretório` onde irá buscar o arquivo `pickle` produzido pelo indexador `mir` e uma lista (possivelmente vazia) com cada `termo` da consulta (conjuntiva). Ao final, deve-se também informar quantos são estes arquivos, como na execução abaixo:

```
[user@local] ./mirs.py -t 30 -r '^.*ãª.*$' diretório primeiro º prêmio
MIR (My Information Retrieval System) de diretório/mir.pickle com 1304 termos e 9 docume
```

```
Palavras que satisfazem a REGEX "^.*ãª.*$"
Total:   1304 Regex:     10 Não regex:   1294 Razão: 129.400 DF >= 1

   DF  Termo/Token     Lista de incidência com IDs dos arquivos
    1  agãªncia       [0]
    1  ciãªncia       [0]
    1  francãªs       [0]
    1  inglãªs        [0]
    1  mãªs           [0]
    1  polonãªs       [0]
    1  portuguãªs     [0]
    1  prãªmio        [0]
    1  prãªmios       [0]
    1  trãªs          [0]
Acima estão os 10 tokens mais frequentes satisfazendo REGEX "^.*ãª.*$". Presentes em 1 a

Conjugação das listas de incidência dos 3 termos seguintes.
   DF  Termo/Token     Lista de incidência com IDs dos arquivos
    4  º               [1, 2, 7, 8]
    5  primeiro        [0, 1, 2, 4, 7]
    5  prêmio          [1, 2, 4, 6, 7]
São 3 os documentos com os 3 termos
1 arq1.txt
2 arq2.txt
7 xarq7.txt
[user@local]
```

Tal análise permite facilmente revelar quais são os arquivos contaminantes. Em nossa coleção de milhares de arquivos-texto, pudemos verificar que algumas dezenas deles tinham o seu encoding erroneamente detectado, de forma que é pedida a implemntação de uma opção de linha de comando `-@ instrucoes` que carrega um arquivo com `instrucoes` ao indexador `mir`. Cada linha do arquivo de `instrucoes` estabelece uma instrução ao indexador. A instrução `@u file\rel\path` prescreve que o *encoding* do arquivo `file\rel\path` será `'utf-8-sig'`. A instrução `@x dir\rel\path` prescreve que o subdiretório `dir\rel\path` será excluído da indexação e a instrução `@x file\rel\path` prescreve que o arquivo `file\rel\path` será igualmente excluído.
Naturalmente que a existência destas prescrições afetam a função `GetFileEncoding`, que será modificada para consultar o dicionário `file\encoding`, como especificado a seguir.

```
def GetFileEncoding(ref_path, file_rel_path):
    """
    Get the encoding of ref_path / file_rel_path using chardet package
    Besides 'encoding' e 'confidence', it sets an extra field: 'errors'.
```

```
    """
    if file_rel_path in file_encoding:
        return file_encoding[ file_rel_path ]
    file_path = os.path.join( raiz, file_rel_path)
    with open(file_path, 'rb') as f:
        file_encoding[ file_rel_path ] = chardet.detect( f.read(MAXSIZE) )
        enc = file_encoding[ file_rel_path ]['encoding']
        size = os.stat( file_path ).st_size
        if size > MAXSIZE and enc=='ascii':
            file_encoding[ file_rel_path ]['encoding'] = 'UTF-8'
            file_encoding[ file_rel_path ]['confidence'] = 0.4
            myerr = 'mixed'
        elif file_encoding[ file_rel_path ]['confidence'] < .63 :
            myerr = 'replace'
        else:
            myerr = 'strict'
        file_encoding[ file_rel_path ]['errors'] = myerr
    return file_encoding[ file_rel_path ]

# Espurious Mixed errors: https://www.i18nqa.com/debug/bug-double-conversion.html
mixed_miscoded_espurious={ b'\x81':1, b'\x8d':1, b'\x90':1, b'\x9d':1, b'\x91':1 }
def mixed_decoder(error: UnicodeDecodeError) -> (str, int):
    global mixed_miscoded_espurious
    """ Trata erros de decodificação Unicode como sendo Windows-1252"""
    bs: bytes = error.object[error.start: error.end]
    if bs in mixed_miscoded_espurious: #ignored
        return '', error.start + 1
    else:
        return bs.decode("Windows-1252"), error.start + 1

    return bs.decode("Windows-1252",errors='ignore'), error.start + 1

import codecs
codecs.register_error("mixed", mixed_decoder)
```

***

Como padrão, não se deve nem fazer *stemming* nem usar *stop lists*.
Estão disponíveis alguns arquivos `.tgz` com estrutura de diretório de arquivos para se testar o sistema e este enunciado poderá sofrer atualizações. O material estará disponível em http:www.ime.usp.br/∼alair/mac0333 .

****

****