Língua Natural

Miniprojecto2 2012/2013

Grupo 5

Ana Santos
annara.snow@gmail.com
nº 56917

Júlio Machado
jules_informan@hotmail.com
nº 57384

Nuno Aniceto
nuno.aja@gmail.com
nº 57682

Breve Sumário

O problema que nos foi proposto semelha-se muito a uma espécie de corrector sintáctico, ou seja, perante uma palavra inserida no terminal que palavras conhecidas pelo sistema se assemelham a ela.

Para a resolução deste problema, nós recorremos ao auxílio de algoritmos como o Índice de Jaccard, Dice e a Distância de Edição (também conhecida como Distância de Levenshtein), sendo estes usados isoladamente ou mesmo de modo combinado.

Descrição da Abordagem seguida

Metodologia de trabalho

Numa primeira abordagem ao problema, nós pensámos em todos os algoritmos e medidas que conhecíamos que permitissem detectar semelhanças entre palavras. Após uma extensa análise decidimos começar pela implementação de Jaccard, Dice e Distância de Edição, embora ficasse sempre em aberto a possível implementação de outras medidas caso o tempo assim o permitisse.

Após escolhidas as medidas para avaliar a semelhança virámos-nos para as ferramentas disponíveis atráves da linguagem de programação JAVA, suas classes e métodos já pré-definidos. Foi nosso objectivo, desde o início não reinventar a roda e portanto todos os métodos e classes definidos usam ao máximo o que a linguagem já oferece.

Após definidas as medidas de semelhança entre palavras, seguiu-se a definição de *thresholds* que iriam indicar ao sistema se tal palavra do sistema tinha ou não um nível de semelhança aceitável ou não.

Os valores para os *thresholds* foram determinados após várias execuções de cada medida (Jaccard, Dice e Distância de Edição) com várias palavras de vários comprimentos.

De modo a melhorar os resultados obtidos, nós resolvemos fazer uma fusão de algumas das medidas.

Arquitectura do módulo

O módulo implementado foi dividido por várias classes, onde a mais importante é a classe Lexer.java, que herda de LexicalTester.java, e que implementa o teste que pretendemos efectuar. As técnicas de medição que escolhemos foram implementadas nas classes MinEditDist.java, Jaccard.java e Dice.java, usando a classe KnownWordNode e OutputNode para tratar do processamento de input e output, respectivamente. A classe NormalizerSimple foi usada para normalizar as palavras passadas no input. Segue-se uma descrição mais detalhada do funcionamento de cada classe

- **Lexer.java**

Nesta classe estão implementadas as principais funcionalidades do módulo ,nomeadamente o tratamento do input, cálculos de distâncias mínimas e coeficientes das técnicas que escolhemos, bom como o tratamento do output e impressão para a consola. Quanto ao tratamento do input, tal como o LexicalTester, o Lexer irá ler um ficheiro de palavras conhecidas, que recebemos da linha de comandos e criar um HashSet contendo todas as palavras conhecidas. Assim o Lexer irá criar um outro HashSet que contém instâncias de KnownWordNode:

```
public Lexer(Set<String> knownWords) {
      super(knownWords);

      this.knownWordNodes = new HashSet<KnownWordNode>(knownWords.size());
      for (String word : knownWords) {
            this.knownWordNodes.add(new KnownWordNode(word));
      }
      /** … remaining code … **/
  }
```

Este HashSet será usado no método test(String word), onde será percorrido, normalizando cada palavra que contém, de modo a poder comparar com a palavra recebida no input, que será também normalizada:

```
@Override
public List<String> test(String word) {
      String normalizedWord = NormalizerSimple.normPunctLCaseDMarks(word);
      ArrayList<String> result = new ArrayList<String>(maximumResultWords);
      ArrayList<OutputNode> evaluationTable = new ArrayList<OutputNode>();

      if(super.getKnownWords().contains(word)) {
            result.add(word);
      } else {
            for (KnownWordNode knownNode : knownWordNodes) {
                  String knownNormalizedWord = knownNode.getNormalizedString();
                  /** … remaining code … **/
            }
            /** … remaining code … **/
      }
  }
```

Após a normalização, as palavras serão comparadas usando os métodos das classes que implementam as técnicas, checkJaccard. checkDice , computeMinimumEditDistance, e computeLevenshteinDistance, de modo a obter valores que nos permitam verificar se a palavra, quando não pertence directamente ao léxico, é semelhante a outras já existentes.

Usámos também uma heurística híbrida, que combina as diferentes técnicas.

Após estes cálculos, se os valores estiverem dentro dos *thresholds* definidos, a palavra conhecida que está a ser comparada com a de input é colocada num OutputNode, juntamente com todos os valores calculados anteriormente. Estes OutputNode vão ser inseridos numa tabela para ordenação.

```
evaluationTable.add(outputNode);
```

De seguida, é feita uma ordenação na ArrayList *evaluationTable*, de modo a que os resultados com valores mais baixos dentro do *threshold,* correspondentes a palavras mais semelhantes à palavra original recebida no input, fiquem no início da ArrayList, uma vez que nos é pedido uma lista de até 6 hipóteses por palavra de input.

Estes valores são inseridos na ArrayList *result*, de onde são impressos para a consola.

- **Classes de Técnicas de Medição de Semelhanças**

As classes MinEditDist, Dice e Jaccard contêm cada uma a implementação dos respectivos algoritmos de distâncias de edição (Levenshtein) e de estatísticas de semelhanças entre conjuntos(Dice e Jaccard), que nos permitem calcular quão semelhantes são as palavras do léxico em relação à palavra lida do input. Será explicado em maior pormenor o funcionamento destes algoritmos e medidas na secção “Motivação, Técnicas e Ferramentas”

- **KnownWordNode**

Esta classe serve essencialmente para conter um HashSet das palavras já conhecidas do léxico de modo a poder normalizá-las e criar uma associação entre a palavra original e a normalizada, para, posteriormente imprimir para a consola a(s) palavra(s) tal como presente(s) originalmente no léxico.

- **OutputNode**

Esta classe tem como função armazenar um KnowWordNode, juntamente com os valores associados às medições e cálculos efectuados, e cujas instâncias serão posteriormente guardadas numa tabela para ordenação na classe Lexer.java.

- **NormalizerSimple**

Esta classe, que nos foi facultada, contém métodos que permitem normalizar as palavras, tanto do input como do léxico, retirando todas as partículas que não interessam, como pontuação, acentuação e marcas diacríticas, de modo a serem mais fáceis de comparar.

Motivação: Técnicas e thresholds definidos

A nossa motivação para usar várias técnicas diferentes prende-se com o facto de ao analisarmos os métodos que tínhamos ao nosso dispor para criar este módulo, verificarmos que nenhum dos métodos garante (como de resto é comum em LN) resultados cem por cento correctos. Mesmo com todas as técnicas que usámos, identificámos algumas falhas, nomeadamente nos métodos mais estatíticos (Dice, Jaccard). Assim não só testámos cada técnica individualmente, bem como criámos uma técnica que tenta usar os valores de todas as técnicas, com pesos adequados, para calcular um valor o mais preciso possível, complementado o valor raso da distância de edição com os valores de probabilidades das técnicas estatísticas.

Como já referido anteriormente, usámos uma técnica que mede a distância de Levenshtein, também conhecida como distância de edição, entre duas palavras. Esta medição é feita contando o número de inserções, substituições e remoções de caracteres necessárias para transformar uma palavra noutra. O algoritmo que usámos (Wagner-Fischer) pega em duas strings, a fonte e o objectivo, e constrói uma matriz (m+1) x (n+1), onde m e n são os comprimentos das strings. Para preencher a matriz são efectuados cálculos simples, cada letra da palavra fonte é comparada com todas as letras da palavra objectivo, e onde estas forem iguais não são necessárias alterações e o valor nessa posição (i , j) mantém se igual ao valor na posição (i-1 , j-1).

Caso sejam diferentes, é escolhido o menor valor de entre as posicões (i-1,j), (i, j-1) e (i-1 , j-1) e é lhe somado 1, o que representa mais uma contagem de uma operação de mudança na palavra. Optámos por usar a distância mínima, onde todos os pesos das variáveis C1, C2 e C3 associadas a este cálculo estão a 1, usando também uma versão onde a variável C3, associada a uma operação de substituição, toma o valor 2 (Levenshtein). Através deste

método, obtemos um número inteiro na última posição da matriz que corresponde ao número mínimo de edições necessárias para se transformar a palavra fonte na palavra objectivo. Basta-nos comparar este número com o threshold para determinar que uma palavra com uma distância de edição superior ao threshold muito provavelmente será muito diferente da palavra recebida no input e portanto não é elegível para sugestão ao utilizador.

Outra das técnicas utilizadas foi o índice de Jaccard, ou coeficiente de semelhança de Jaccard, uma medida estatística usada para comparar semelhanças e diversidade em conjuntos de amostras. Esta medida é definida como o tamanho da intersecção dividida pelo tamanho da união do conjunto de amostras, no nosso caso a as palavras em teste. O algoritmo que implementámos toma duas strings como input e calcula a união entre as duas, que é uma string contendo todos os caracteres de ambas as palavras recebidas, sem repetições. Calcula de seguida a intersecção, todos os caracteres em comum entre as duas string e usa o tamanho das strings assim obtidas para efectuar o cálculo do indice, usando o método checkJaccard:

```
public double checkJaccard() {
	double i = intersection();
	double j = union();
	return i/j;
}
```

onde *intersection()* e *union()* calculam os respectivos tamanhos.

Uma vez que obtemos valores entre 0 e 1, quanto mais próximo de 1 estiver o valor, maior é a semelhança entre palavras, logo maior a probabilidade de a palavra constituir uma sugestão para apresentar ao utilizador.

A terceira e última técnica utilizada foi o coeficiente de Dice, que é também uma medida estatística de semelhança de conjuntos. Embora seja semelhante a Jaccard, usa uma fórmula ligeiramente diferente:

```
public double checkDice() {
	double i = intersection();
	double x = _str1.size();
	double y = _str2.size();
	return (2*i)/(x+y);
}
```

Como mostrado no método *checkDice(),* a fórmula usa o dobro do tamanho da intersecção entre as duas strings a dividir pela soma do tamanho das duas strings.

Tal como Jaccard iremos obter um valor entre 0 e 1, de onde podemos concluir o mesmo que acima: valores mais próximos de 1 indicam uma maior semelhança entre palavras e consequentemente maiores hipóteses de escolha como sugestão.

De notar que esta semelhança pode ser calculada usando bigramas, segundo a fórmula

$$s = \frac{2n_t}{n_x + n_y}$$

onde nt é o número de bigramas em ambas as strings e nx e ny o número de bigramas das strings x e y respectivamente.

A técnica híbrida que desenvolvemos consiste numa junção das técnicas usadas anteriormente, de modo a obter tentativamente um valor médio consistente, que nos permita uma melhor correspondência de semelhanças entre as palavras de input e as palavras do léxico. Foi usada para este efeito a seguinte fórmula heurística,

```
	float heuristic = 0;
	heuristic += JaccardIndexWeight*JaccardValue;
	heuristic += DiceCoefficientWeight*DiceValue;

	heuristic /= (minimumEditDistanceWeight*MinimumEditDistanceValue
		    + LevenshteinDistanceWeight*LevenshteinDistanceValue);
```

o que corresponde essencialmente a usar Jaccard e Dice como factores positivos, uma vez que estes nos dão a probabilidade de semelhança, contrabalançados com as distâncias de edição, onde assumimos que quanto maiores forem, mais operações de mudança e portanto menor a semelhança entre as palavras.

Como temos a possibilidade de qualquer combinação de testes, há a hipótese de não se usar nem o teste ao índice de Jaccard ou ao coeficiente de Dice e nesse caso (para não ter um valor de heurística infinito) assumimos que a heurística é o simétrico da distância calculada – seja da distância mínima de edição, da distância de Levenshtein ou de uma combinação entre estas.

```
if(testMinimumEditDistance)
      heuristic = -(minimumEditDistanceWeight*MinimumEditDistanceValue);
if(testLevenshteinDistance)
      heuristic = -(LevenshteinDistanceWeight*LevenshteinDistanceValue);
```

Os thresholds são um componente crítico do nosso módulo, visto que são estes que determinam se uma palavra do léxico é elegível para ser mostrada como sugestão ao utilizador, e como consequência, thresholds demasiado altos fariam aceitar a maioria das palavras, e inversamente, demasiado baixos rejeitariam quase todas as palavras. Através de testes e ajustamentos nos valores, chegámos aos valores base que usamos no módulo.Contudo, estes thresholds podem ser alterados via um ficheiro de configuração, o que permite a quem utilizar o módulo usar valores que lhes sejam mais convenientes. Por vezes, diferenças de uma unidade nos coeficientes entre testes resultavam em diferenças imediatamente aparentes.

Exemplo:
Com threshold MED = 4
palavra construíram: construir, construída, construção, construído, construiu,
Com threshold MED = 5
palavra construíram: construir, construída, construído, construiu, consta.

Tentámos assim ajustar os valores que nos dessem uma maior percentagem de palavras semelhantes á palavra lida do input, o que conjugado com a heurística híbrida

Em termos de ferramentas, para além de tudo o que temos disponível na linguagem Java, foi-nos facultada a classe NormalizerSimple, que já descrevemos na secção “Arquitectura do Módulo”, a qual usámos para retirar caracteres que não interessam, facilitando imenso o processo de comparação de strings caractere a caractere.

Avaliação

Os testes encontram-se descritos em anexo.
São feitos testes individuais a cada técnica, e no final um teste geral que faz uso de todas as técnicas.

Conclusões e trabalho futuro

Problemas encontrados

Verificámos que tanto Jaccard quanto Dice tinham o mesmo problema, por mais que se alterásse o valor dos *thresholds*, uma ou outra palavra não desejada aparecia nos resultados. A razão para tal acontecer devia-se ao facto que ambos trabalham sobre conjuntos de letras, união e intersecção de letras. Portanto numa comparação ente a palavra “bibliotecas” e a palavra “castelo” o Dice, a intersecção do conjunto de letras de ambas palavras vai dar a [c,a,s,t,e,l,o] e um resultado de 0.875 de similaridade, ou seja 0.125 de erro. Uma margem muito baixa.

Tarefas que ficaram por fazer

Por falta de tempo, nós acabámos por não implementar a medida de Jaro, que dá grande ênfase ao comprimento de prefixos reduzindo o grau de similaridade quando as palavras em questão não partilham o mesmo prefixo (podendo este tomar um comprimento variável de 1 ao tamanho da palavra mais pequena). Há também a hipótese de usar variações na distância mínima de edição para contar separadamente: o número de inserções, o número de eliminações, e o número de substituições – e desse modo, com vários thresholds, poderia-se atribuir mais informação heurística levando a um resultado mais ‘controlado’.

Referências

[1] Levenshtein Distance [wikipedia]
[2] Wagner-Fischer Algorithm [wikipedia]
[3] Jaccard Index [wikipedia]
[4] Dice’s Coefficient [wikipedia]
[5] Jaro’s Distance [scribd]

Aqui mostramos resultados de testes individuais a cada técnica e ainda um teste que combina todas as técnicas.

Valores de thresholds e pesos usados nos testes:

JaccardIndexThreshold	0.5
DiceCoefficientThreshold	0.75
minimumEditDistanceThreshold	4
LevenshteinDistanceThreshold	6

Todas as técnicas são combinadas com o mesmo peso, de 1.0

Sequência de palavras testadas: “aeiou toto bola viv construíram Biliotecas actua mente”
Note que a heurística é mostrada com a palavra-chave “verboseMode” no ficheiro de configuração.

JACCARD

For the word 'aeiou' found the following candidates:
- [h: 0,66667] leilão
- [h: 0,62500] arquiteto
- [h: 0,57143] reabriu
- [h: 0,57143] queria
- [h: 0,57143] leiloado

For the word 'toto' found the following candidates:
- [h: 0,66667] todo
- [h: 0,66667] totoloto
- [h: 0,66667] tão
- [h: 0,50000] sítios
- [h: 0,50000] tipo

For the word 'bola' found the following candidates:
- [h: 0,80000] totobola
- [h: 0,75000] boa
- [h: 0,75000] olá
- [h: 0,60000] local
- [h: 0,60000] boas

For the word 'viv' found the following candidates:
- [h: 0,66667] vivia
- [h: 0,66667] vai
- [h: 0,66667] vive
- [h: 0,66667] vivo
- [h: 0,50000] vivem

For the word 'construíram' found the following candidates:
- [h: 0,81818] construída
- [h: 0,80000] construção
- [h: 0,80000] romantismo
- [h: 0,80000] construir
- [h: 0,80000] construiu

For the word 'Biliotecas' found the following candidates:
- [h: 0,88889] biblioteca
- [h: 0,80000] instalações
- [h: 0,77778] castelo
- [h: 0,77778] estilísticas
- [h: 0,70000] instalação

For the word 'actua' found the following candidates:
- [h: 0,80000] custa
- [h: 0,80000] actual
- [h: 0,80000] tchau
- [h: 0,75000] tua
- [h: 0,66667] executa

For the word 'mente' found the following candidates:
- [h: 0,80000] monte
- [h: 0,75000] tem
- [h: 0,60000] neste
- [h: 0,60000] tens
- [h: 0,60000] nome

DICE

For the word 'aeiou' found the following candidates:
- [h: 0,80000] leilão
- [h: 0,76923] arquiteto

For the word 'toto' found the following candidates:
- [h: 0,80000] totoloto
- [h: 0,80000] tão
- [h: 0,80000] todo

For the word 'bola' found the following candidates:
- [h: 0,88889] totobola
- [h: 0,85714] boa
- [h: 0,85714] olá
- [h: 0,75000] boas
- [h: 0,75000] cabo

For the word 'viv' found the following candidates:
- [h: 0,80000] vivia
- [h: 0,80000] vai
- [h: 0,80000] vive
- [h: 0,80000] vivo

For the word 'construíram' found the following candidates:
- [h: 0,90000] construída
- [h: 0,88889] romantismo
- [h: 0,88889] construiu
- [h: 0,88889] construção
- [h: 0,88889] construer

For the word 'Biliotecas' found the following candidates:
- [h: 0,94118] biblioteca
- [h: 0,88889] instalações
- [h: 0,87500] estilísticas
- [h: 0,87500] castelo
- [h: 0,82353] ecletismo

For the word 'actua' found the following candidates:
- [h: 0,88889] actual
- [h: 0,88889] tchau
- [h: 0,88889] custa
- [h: 0,85714] tua
- [h: 0,80000] executa

For the word 'mente' found the following candidates:
- [h: 0,88889] monte
- [h: 0,85714] tem
- [h: 0,75000] neste
- [h: 0,75000] nome
- [h: 0,75000] tens

DISTÂNCIA MÍNIMA DE EDIÇÃO

For the word 'aeiou' found the following candidates:
- [h: -2,0] criou
- [h: -2,0] levou
- [h: -3,0] feitos
- [h: -3,0] feito
- [h: -3,0] you

For the word 'toto' found the following candidates:
- [h: -1,0] todo
- [h: -2,0] porto
- [h: -2,0] tão
- [h: -2,0] todos
- [h: -2,0] tipo

For the word 'bola' found the following candidates:
- [h: -1,0] boa
- [h: -1,0] olá
- [h: -2,0] bom
- [h: -2,0] boas
- [h: -2,0] lá

For the word 'viv' found the following candidates:
- [h: -1,0] vive
- [h: -1,0] vivo
- [h: -2,0] viuvo
- [h: -2,0] dia
- [h: -2,0] vivia

For the word 'construíram' found the following candidates:
- [h: -2,0] construída
- [h: -2,0] construir
- [h: -3,0] construído
- [h: -3,0] construiu
- [h: -3,0] construção

For the word 'Biliotecas' found the following candidates:
- [h: -2,0] biblioteca

For the word 'actua' found the following candidates:
- [h: -1,0] actual
- [h: -2,0] altura
- [h: -2,0] acaba
- [h: -2,0] tua
- [h: -3,0] lua

For the word 'mente' found the following candidates:
- [h: -1,0] monte
- [h: -2,0] neste
- [h: -2,0] deste
- [h: -2,0] este
- [h: -3,0] teve

LEVENSHTEIN

For the word 'aeiou' found the following candidates:
- [h: -3,0] ou
- [h: -3,0] ao
- [h: -3,0] eu
- [h: -4,0] o
- [h: -4,0] feito

For the word 'toto' found the following candidates:
- [h: -2,0] todo
- [h: -3,0] porto
- [h: -3,0] o
- [h: -3,0] tão
- [h: -3,0] todos

For the word 'bola' found the following candidates:
- [h: -1,0] boa
- [h: -1,0] olá
- [h: -2,0] boas
- [h: -2,0] lá
- [h: -3,0] bom

For the word 'viv' found the following candidates:
- [h: -1,0] vive
- [h: -1,0] vivo
- [h: -2,0] viuvo
- [h: -2,0] vivia
- [h: -2,0] vivem

For the word 'construíram' found the following candidates:
- [h: -2,0] construir
- [h: -3,0] construída
- [h: -4,0] construiu
- [h: -5,0] construído
- [h: -5,0] construção

For the word 'Biliotecas' found the following candidates:
- [h: -2,0] biblioteca
- [h: -6,0] boas

For the word 'actua' found the following candidates:
- [h: -1,0] actual
- [h: -2,0] tua
- [h: -3,0] cá
- [h: -3,0] altura
- [h: -3,0] tu

For the word 'mente' found the following candidates:
- [h: -2,0] monte
- [h: -3,0] me
- [h: -3,0] este
- [h: -3,0] te
- [h: -4,0] neste

COMBINAÇÃO DAS QUATRO TÉCNICAS

For the word 'aeiou' found the following candidates:
- [h: 0,16296] leilão

For the word 'toto' found the following candidates:
- [h: 0,48889] todo
- [h: 0,29333] tão
- [h: 0,18333] totoloto

For the word 'bola' found the following candidates:
- [h: 0,80357] boa
- [h: 0,80357] olá
- [h: 0,33750] boas
- [h: 0,21111] totobola
- [h: 0,16875] cabo

For the word 'viv' found the following candidates:
- [h: 0,73333] vive
- [h: 0,73333] vivo
- [h: 0,36667] vivia
- [h: 0,36667] vai

For the word 'construíram' found the following candidates:
- [h: 0,42222] construir
- [h: 0,34364] construída
- [h: 0,24127] construiu
- [h: 0,21111] construção
- [h: 0,19617] construído

For the word 'Biliotecas' found the following candidates:
- [h: 0,45752] biblioteca

For the word 'actua' found the following candidates:
- [h: 0,84444] actual
- [h: 0,40179] tua
- [h: 0,22500] tuas
- [h: 0,21111] custa
- [h: 0,16889] tchau

For the word 'mente' found the following candidates:
- [h: 0,56296] monte
- [h: 0,22500] neste
- [h: 0,20089] tem
- [h: 0,16875] tens
- [h: 0,15000] nome