

|  |
| --- |
| Trabalho pratico de AED |
|  |
| 15 de maio  ISMAT  Criado por:  Cayan Prola e Diogo Fernandes |



# Descrição da tarefa

|  |
| --- |
| Foi nos disposta a tarefa de criar um programa no qual lê os dados de diferentes data sets, guarda-os em um hash map e em sequencia guarda esse hash map em um arraylist de hash maps.  Com estes dados, trabalhamo-los de acordo com os argumentos passados no programa. Os argumentos definem quais arquivos serão usados, quais as colunas dos arquivos serão selecionadas e qual a métrica a ser utilizada. As métricas feitas foram Levenshtein, Damerau-Levenshtein, Jaccard, Hamming e Cosine. Cada uma com seu melhor caso de uso. |

Processo de desenvolvimento

Inicialmente, focamos em entender o problema e qual a nossa tarefa, a seguir a isso dividimos a tarefa em partes e começamos a desenvolver o projeto.

No primeiro passo, precisamos ler diversos data sets no formato ".csv" e guardar os dados de cada data set em um hash map, que seguidamente será guardado em um arraylist de hashmaps. Utilizamos o scanner e a classe file para isso. Para trabalharmos melhor os dados, dividimos cada coluna usando uma variável chamada de header, e assim selecionamos apenas as colunas que queremos utilizando o header escolhido. Apos termos todos os dados devidamente salvos, começamos a criar as classes para as métricas de similaridade de Levenshtein, Jaccard, Cosine, Hamming e Damerau-Levenshtein. Usamos uma classe abstrata chamada “metrics” para ser a classe parente de todas as nossas métricas. Fomos criando métrica por métrica e testando com todos os data sets e headers diferentes para entendermos melhor como lidarmos com os dados e achar alguma irregularidade. Notamos que para data sets muito grandes, tivemos alguns problemas com os resultados, em algumas situações apenas alguns valores foram escritos no console, possivelmente por alguma configuração das nossa IDEs.

# Levenshtein

Primeiramente foi criada a classe para Levenshtein, na qual é usada principalmente para a comparação entre palavras. É feita a seleção das 2 colunas que queremos comparar nos parâmetros do método “calculate”, no qual chamamos o método “calcDistance” para calcular a distância entre as duas palavras adicionando-as a um array bidimensional chamado "distance". Em seguida usamos uma comparação ternária para salvar o custo de mudar o char de uma string para ser igual o char da outra. Comparando todos os seus chars e utilizando o custo para mudar as strings, pegamos o menor valor e o retornamos como nossa distância. Assim no método “calculate” efetuamos o cálculo da distância e logo apos utilizamos a fórmula:

(str1.length() + str2.length() - distance) / (str1.length() + str2.length()); para efetuar o cálculo da similaridade entre cada palavra, e efetuamos o print dos resultados.

A picture containing text, screenshot

Description automatically generated

# Jaccard

Apos terminar a classe de Levenshtein, fomos para a classe de Jaccard, na qual é mais bem utilizada para comparar frases.

Considerando isso, utilizamos os valores de dois mapas, efetuamos a intersecção entre os dois, logo apos calculamos a união entre ambos e utilizamos a fórmula: intersecção / união; e damos print no resultado. Chegando assim ao resultado esperado.

A screen shot of a computer program

Description automatically generated with low confidence

# Hamming

Para a métrica de Hamming, ambas strings precisam ter o mesmo tamanho para serem comparadas, caso não tenham, podem ser tomadas duas decisões: adicionar espaços na string menor até ficar do tamanho da string maior, ou não efetuar a comparação. Decidimos ir para o segundo caso, que seria o melhor uso para esta métrica. Portanto comparamos o tamanho das strings, caso iguais, calculamos a distância (chars diferentes) entre cada string e a retornamos no método "calcDistance". Em seguida utilizamos o método “calculate”, que utiliza a biblioteca “Math” para salvar qual o maior comprimento entre as duas strings e o salva em uma variável chamada “maxLength”, apos utilizamos a fórmula: maxLength - distance / maxLength. Sendo assim escrevendo a similaridade entre as strings.

A screenshot of a computer program

Description automatically generated with medium confidence

# Cosine

Cosine é utilizada para comparar dois vetores em um espaço vetorial. Como temos texto como nossos valores, necessitamos de converter para vetores numéricos para efetuar os cálculos, apos isso precisamos de ambos os valores de magnitude, e o produto de ambos vetores para efetuarmos o cálculo, recebemos a magnitude de ambos os mapas com o metodo “calculateMagnitude” e efetuamos o calculo do produto de ambos na variavel “dotProduct”, aplicamos a formula: dotProduct / (magnitude1 \* magnitude2); para recebermos o resultado final.

A picture containing text, screenshot

Description automatically generated

A screen shot of a computer program

Description automatically generated with medium confidence

# Damerau-Levenshtein

Diferentemente da similaridade de Levenshtein, aqui podemos também transpor caracteres adjacentes, algo que Levenshtein não permite, portanto para chegarmos ao resultado desejado precisamos considerar posições diferentes na comparação de chars.

Primeiro utilizamos um array bidimensional para receber a distância de todas as strings, e depois utilizamos uma operação ternária para calcular o custo das operações, escolher a melhor operação possível e checar se ela é valida, caso seja a tabela é alterada de acordo com a operação e é retornado o valor com as novas distâncias.

Em seguida, iteramos sobre todas os pares de strings e calculamos a sua similaridade utilizando a fórmula: (-1.0 + (distanceSum / (double) totalCount)); assim já escrevendo o resultado no console.

A screenshot of a computer program

Description automatically generated with medium confidence

A picture containing text, screenshot, font

Description automatically generated

# Alguns resultados obtidos

Argumentos:

fuzzy-join --filename1=names1.csv --filename2=names2.csv --name1=location --name2=Location --distance=cosine

A screen shot of a computer

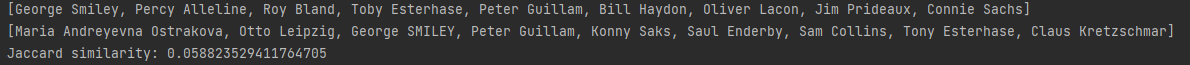
Description automatically generated with medium confidence

fuzzy-join --filename1=names1.csv --filename2=names2.csv --name1=location --name2=Location --distance=Levenshtein

A screenshot of a computer

Description automatically generated

fuzzy-join --filename1=names1.csv --filename2=names2.csv --name1=name --name2="Person Name" --distance=jaccard



fuzzy-join --filename1=names1.csv --filename2=names2.csv --name1=name --name2="Person Name" --distance=hamming

A screenshot of a computer

Description automatically generated with medium confidence

fuzzy-join --filename1=gpt.csv --filename2=population.csv --name1=country --name2=country --distance=damerau-levenshtein

A screenshot of a computer

Description automatically generated

# Conclusão

Apos trabalharmos com diferentes métricas e grupos de dados, nota-se a necessidade de algoritmos bem estruturados para a solução de dados problemas, como o algoritmo necessário para efetuar o cálculo da similaridade de Cosine. Foram usadas técnicas aprendidas durante a cadeira de Algoritmia e Estrutura de Dados, como a programação dinâmica, entre outros casos. Apos o trabalho podemos notar ainda mais a importância de algoritmos e o trabalho em que os dados devem sofrer para serem utilizados da melhor maneira.