Universidade Federal do Pampa

Ciência da Computação

Aprendizado de Máquina

Relatório de Trabalho Prático

Alunos:
Gustavo C. Rodrigues
Wolgan E. Quepfert

Professor:
Dr. Marcelo R. Thielo

18 de julho de 2018

1 Introdução

A tarefa de agrupamento ou *clustering*, é uma técnica de mineração de dados que tem como objetivo tentar criar grupos ou *clusters* de objetos (dados, informações, etc) que potencialmente se relacionam de alguma forma. Em aprendizado de máquina, este processo é um método aplicado ao grupo de problemas de aprendizado não supervisionado, através de métodos numéricos a partir de informações presentes nas variáveis de cada caso. Esta técnica tem aplicabilidade em diversas tópicos de conhecimento, como por exemplo classificação de documentos textuais, reconhecimento de padrões, análise de dados espaciais, agrupamento de pacientes médicos por sintomas, etc. A Figura 1 mostra um exemplo da aplicação de notícias da Google que agrupa notícias com base em categorias de notícias semelhantes.

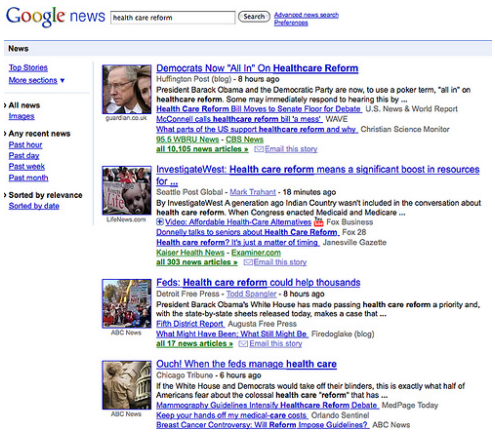

Figura 1: Google News

Para realizar a tarefa de agrupamento ou clusterização, diversos algoritmos vem sendo desenvolvidos e aprimorados ao longo do tempo, onde um dos principais *trade-offs* para a escolha, está na necessidade da obtenção de soluções de resultados com uma qualidade e tempo de resposta razoáveis. Dentre os algoritmos mais populares estão: K-Means, K-medoids, CLARA, CLARANS e algoritmos genéticos. Neste contexto, este relatório visa reportar a aplicação de um algoritmo genético para agrupar uma coleção de frases publicadas na rede social Tweeter com base na similaridade de seus conteúdos.

1.1 Objetivo

Implementar um algoritmo genético que execute a tarefa de agrupar *tweets* considerando a similaridade de conteúdo entre os mesmos. Para atingir esse objetivo é necessário:

1. Ler e tratar os textos de entrada
2. Definir métrica de similaridade entre os *tweets*
3. Definir o algoritmo genético aplicado ao problema
4. Gerar e analisar os resultados

2 Algoritmos Genéticos

Algoritmos Evolucionários ou Algoritmos Genéticos (AG) são métodos de busca e otimização que possuem inspiração nos conceitos da teoria da seleção natural das espécies, proposta por Charles Darwin.

Estes algoritmos são inspirados nos processos genéticos de organismos biológicos para procurar soluções ótimas ou sub-ótimas para problemas complexos ou com espaço de soluções muito grande. Para tanto, procede-se da seguinte maneira: codifica-se cada possível solução de um problema em uma estrutura chamada de "cromossomo" (individuo), que é composta por uma cadeia de bits. Estes indivíduos "evoluem" ao longo de várias gerações, de forma similar aos seres vivos de acordo com os princípios da seleção natural e sobrevivência dos mais aptos.

Os indivíduos são submetidos a um processo evolucionário que envolve avaliação, seleção, recombinação e mutação. Após varias gerações ou ciclos de evolução, a população deverá conter indivíduos mais aptos. Os AG utilizam uma analogia direta deste processo evolutivo, onde cada indivíduo apresenta uma possível solução para um problema dado. A cada cromossomo, atribui-se um valor de avaliação, que indica o quanto a solução representada por este individuo é boa em relação às outras soluções daquela geração.

Geração ou população refere-se ao conjunto de todas as soluções com as quais trabalha o sistema. Aos indivíduos com maior aptidão é dada uma probabilidade de se "reproduzirem" e passarem seus "genes" adiante. A mutação

tem um papel significativo, ao introduzir na população novos indivíduos gerados modificando indivíduos da população de maneira aleatória. Para determinar o final da evolução pode-se definir um número máximo de gerações, número de indivíduos criados ou adicionar um algoritmo para classificação de solução satisfatória. A Figura 2 apresenta um possível pseudocódigo de um algoritmo genético.

```
inicializaPopulacao()
calculaAptidao()
while not criterioDeParada:
    selecao()
    cruzamento()
    mutacao()
    calculaAptidao()
    substituiPopulacao()
```

Figura 2: Pseudcódigo de um algoritmo genético

3 Desenvolvimento

Para o desenvolvimento do algoritmo evolutivo aplicado ao problema de clusterização, precisamos definir os seguintes itens:

1. Linguagem de programação
2. Definição da matriz de dissimilaridade entre o *tweets*
3. Representação das entidades componentes do problema
4. Função para avaliação das soluções

A linguagem de programação Python foi escolhida tendo em vista que os membros do grupo já tiveram contato com a mesma e esta é amplamente utilizada para resolução de problemas em Aprendizado de Máquina, com uma comunidade fortemente ativa que dispõe de diversas bibliotecas que auxiliam na execução das etapas necessárias para construir o algoritmo.

Para representar os indivíduos, uma das entidades mais fundamentais de um algoritmo genético, foi utilizada uma matriz que basicamente define a qual *cluster* um determinado *tweet* pertence, onde linhas representam os *tweets* e colunas os *clusters*, dessa forma para um *tweet* i, a coluna k que estiver com o valor 1, indica que o *tweet* i pertence ao *cluster* k, do contrário a coluna recebe o valor 0.

3.1 Matriz de Dissimilaridade

Para que seja possível avaliar as soluções geradas pelo algoritmo, precisamos definir uma matriz que contém o valor da dissimilaridade entre todos os *tweets*, tendo em vista que o processo de avaliação depende de consultas à essa matriz. Visando analisar a similaridade de conteúdos textuais, podemos fazer uma representação vetorial dos *tweets*, onde as componentes dos vetores são representadas pela importância das palavras que compõe o conteúdo dos mesmos.

Para calcular a relevância de um termo na coleção de *tweets*, utilizamos a métrica TF-IDF. Com essa representação vetorial, a medida de similaridade entre os mesmos pode ser obtida através da técnica *cosine similarity*.

TF-IDF, acrônimo de *Term Frequency - Inverse Document Frequency*, é uma técnica derivada das áreas de mineração de texto e recuperação de informação, utilizada para mensurar a importância de uma palavra dentro de uma coleção ou corpus de documentos. Assumindo que a importância de uma palavra está relacionada com o número de vezes que a mesma ocorre, o valor de TF-IDF aumenta proporcionalmente ao número de vezes que uma palavra aparece em um documento e tem seu valor ajustado de acordo com a frequência da palavra nos demais documentos do corpus, prevenindo com que palavras muito frequentes (não tão importantes) em todos os documentos interfiram na sua precisão. A frequência de uma palavra em um documento (tf) é calculada pela Equação 1. Seja d um documento qualquer e t um termo ou palavra qualquer pertencente ao documento d:

$$tf(t,d) = \frac{f_d(t)}{w \in d \max f_d(w)} \tag{1}$$

Onde $f_d(t)$ retorna a frequência da palavra t no documento d e $w \in d \max f_d(w)$ retorna a maior frequência entre as demais palavras w no documento d. Como documentos mais extensos tendem a conter uma maior frequência de palavras, esta divisão é feita a fim de normalizar a frequência das palavras com o tamanho do documento.

O cálculo da frequência inversa do documento (do inglês, *Inverse Document Frequency*, ou IDF) é realizado pela Equação 2 da seguinte forma: seja t um termo ou palavra qualquer e D uma coleção ou corpus de documentos.

$$idf(t,D) = \ln\left(\frac{|D|}{|d \in D : t \in d|}\right) \tag{2}$$

O número total de documentos é dividido pelo número de documentos em D que possuem o termo ou palavra t, e por fim, aplica-se a função logarítmica para alterar a escala do valor. O resultado final ($tfidf$, Equação 3) é obtido multiplicando o valor de tf (Equação 1) por idf (Equação 2).

$$tfidf(t, d, D) = tf(t, d).idf(t, D) \tag{3}$$

A medida de similaridade entre vetores conhecida como *cosine similiarity*, é dada através do cálculo do produto interno dos mesmos. Ao representarmos informações no espaço vetorial, podemos utilizar o cosseno do ângulo formado entre eles para comparar suas orientações. O cosseno de 0 é 1, ou seja, se o ângulo formado entre dois vetores é igual a 0 (paralelos) então eles são avaliados como "idênticos", qualquer outro grau formado entre os vetores, resulta em um valor menor do que 1.

Dois vetores ortogonais tem similaridade 0 enquanto que vetores opostos tem similaridade igual a -1. Como apenas a orientação dos vetores é levada em consideração, fica claro que para esta medida, a magnitude dos vetores é irrelevante para o cálculo. Dado dois vetores de dimensão N, v e w, a similaridade entre eles pode ser calculada pela Equação 4.

$$cosinSim(\vec{v}, \vec{w}) = \frac{\vec{v} \bullet \vec{w}}{|\vec{v}|\,|\vec{w}|} = \frac{\sum_{i=1}^{N} v_i \times w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{4}$$

A matriz de dissimilaridade D foi obtida com o auxílio da biblioteca sklearn que gera a representação vetorial dos *tweets*, como pode ser visto na Figura 3

```
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(tweets)
d = pairwise_distances(X=tfidf_matrix, Y=None, n_jobs=-1, metric
    ='cosine')
```

Figura 3: Matriz de dissimilaridade.

Como apresentado na Figura 2, diversos métodos precisam ser definidos na construção de um algoritmo genético. As próximas subseções, apresentam os principais trechos de código da implementação desses métodos no presente trabalho.

3.2 Inicialização aleatória

Para a solução proposta, o tamanho da população foi definido em 150 visando uma maior variedade de indivíduos a disposição, dessa forma, precisamos gerar 150 indivíduos e definir seu *membership* a um determinado *cluster* de maneira aleatória. A Figura 4 mostra a parte principal da função de inicialização aleatória da população, onde o índice da coluna (*cluster*) de um indivíduo é definida "ao acaso".

```
def random_start(self):
    for i in range(self.n_tweets):
        cluster_index = randint(0, self.n_clusters - 1)
        membership_matrix[i][cluster_index] = 1
```

Figura 4: População inicial aleatória.

3.3 Cálculo de Aptidão

A aptidão de um indivíduo é responsável por indicar a qualidade dos *clusters* gerados com base na soma das dissimilaridades médias de seus membros, a equação utilizada para o cálculo da aptidão pode ser vista na Equação 5.

$$\varepsilon_k^{pc}(M) = \sum_{v=1}^{K} \frac{1}{2p_v N} \sum_{k=1}^{N} \sum_{l=1}^{N} M_{kv} M_{lv} D_{kl}$$
$$\text{Onde} \quad p_v = \sum_{k=1}^{N} \frac{M_{kv}}{N} \tag{5}$$

Sendo K o número de *clusters* pré-determinado, N o número total de tweets a serem classificados, $D \in \Re^{NxN}$ a matriz de proximidade (ou dissimilaridade) entre cada elemento, e $M \in \{0,1\}^{N \times K}$ é a matriz de associação que será manipulada pelo algoritmo a fim de minimizar a função de custo/aptidão $\varepsilon(M)$.

3.4 Seleção

Para selecionar um par de indivíduos da população que combinados resultarão em outros dois indivíduos, foi utilizado o algoritmo de torneio que a

partir de uma amostra de tamanho k da população, seleciona os 2 mais aptos (vide Cálculo de Aptidão). A Figura 5 da uma visão geral do método.

```
def selection(self, k):
    lista = sample(self.population, k)
    lista.sort(key = lambda x : x.fitness)
    return lista[0]
```

Figura 5: Seleção de k indíviduos para cruzamento via torneio.

3.5 Cruzamento

O cruzamento ou *Crossover* foi realizado através do algoritmo de combinação por um ponto, onde a partir de um corte nos dois indivíduos obtidos na seleção, combinamos a parte superior com a inferior de cada um deles, gerando assim dois novos indivíduos com as características dos "pais". Como pode ser visto na Figura 6 o índice de corte é escolhido randomicamente.

```
def crossover(self, parents):
    a, b = parents
    cutt_index = randint(int(self.n_tweets * 0.1), self.n_tweets
        -1)

    top_a = copy.deepcopy(a.membership_matrix[0:cutt_index])
    bottom_a = copy.deepcopy(a.membership_matrix[cutt_index:])

    top_b = copy.deepcopy(b.membership_matrix[0:cutt_index])
    bottom_b = copy.deepcopy(b.membership_matrix[cutt_index:])

    new_membership_matrix_a = top_a + bottom_b
    new_membership_matrix_b = top_b + bottom_a

    son_a = Individual(new_membership_matrix_a, self.generation)
    son_b = Individual(new_membership_matrix_b, self.generation)

        return (son_a, son_b)
```

Figura 6: Cruzamento de dois indivíduos a e b.

3.6 Mutação

A probabilidade de uma mutação ocorrer foi pré-definida em 1%, visando auxiliar no processo convergência sem realizar alterações desnecessárias. A

Figura 7 mostra a implementação utilizada para a mutação, onde o *tweet* é removido do seu antigo *cluster* e inserido em um novo de maneira aleatória.

```
def mutation(self, individual):
    mutation_rate = random()
    if mutation_rate <= 0.01:
        new_cluster_index = randint(0, self.n_clusters - 1)
        row[row.index(1)] = 0
        row[new_cluster_index] = 1
```

Figura 7: Mutação de um indivíduo.

3.7 Substituição da População

Novas gerações são criadas repetindo os processos definidos anteriormente até que o número de indivíduos da nova geração seja o pré-definido (150 em nosso experimento). Quando uma nova geração está com completa, ou seja, com todos os indivíduos gerados, ela substitui a população antiga como pode ser visto na Figura 8.

```
def new_population(self):
    aux_population = list()
    while not self.complete_population(aux_population):
        a, b = self.crossover((parent_a, parent_b))
        aux_population.append(a)
        aux_population.append(b)
    self.population = aux_population
        self.iterations += 1
        self.generation += 1
```

Figura 8: Substituição da população.

3.8 Elitismo

Uma operação adicional que é comumente realizada em algoritmos genéticos é conhecida como elitismo. Esta operação substitui o pior indivíduo da nova geração, pelo melhor indivíduo da geração anterior, visando não perder boas soluções candidatas no processo estocástico do algoritmo. Na Figura 9 vemos a implementação desse processo.

```
best_from_old = min(self.population, key=lambda ind: ind.fitness
    )
aux_population.remove(max(aux_population, key=lambda ind: ind.
    fitness))
```

```
aux_population.append(best_from_old)
```

Figura 9: Elitismo.

3.9 Critério de Parada

Para determinar quando a execução do algoritmo deve ser encerrada, utilizamos um critério baseado no número de iterações em que a aptidão do melhor indivíduo até o momento não diminui. O contador de iterações é reiniciado toda vez que um indivíduo com melhor aptidão é gerado. Adicionalmente um número máximo de gerações pode ser definido para obrigar o algoritmo a interromper sua execução como pode ser visto na Figura 10

```
def satisfaction(self):
    return self.iterations < self.max_iterations or self.
        generation == 10000
```

Figura 10: Critério de parada do algoritmo.

4 Resultados

No experimento, foram utilizados 251 *tweets* como entrada e geramos soluções para 5, 10 e 25 *clusters*, que são então armazenados em arquivos para análise posterior. Esta seção apresenta os resultados obtidos da execução do algoritmo genético desenvolvido e algumas discussões sobre esses resultados.

Para diferentes números de *clusters*, é esperado que haja uma discrepância entre o custo das melhores soluções geradas. Mais especificamente, quanto maior o número de *clusters*, melhor tendem a ser os resultados. Na Tabela 1 podemos verificar o resultado das melhores soluções obtidas para os 3 tamanhos de *clusters*.

# *Clusters*	Custo da melhor solução
5	96.5
10	69.0
25	36.0

Tabela 1: Melhor custo obtido para a 5, 10 e 25 *clusters*.

Esta diferença é algo esperado, uma vez que existem exatamente 25 tipos de *tweets* na entrada com pequenas variações em seu conteúdo. Dessa forma,

quanto menor o número de *clusters*, maior o número de tipos de texto serão forçados a ficar no mesmo *cluster*, aumentando a dissimilaridade intra-*cluster*.

Na solução gerada para 5 *clusters*, o número de *tweets* atribuídos a cada um deles é respectivamente: 39, 18, 61, 70, 63. O que se destaca aqui, é a quantidade bastante inferior de objetos no *cluster* 2 (18). Uma possível explicação pode estar relacionada ao fato desse *cluster* possuir *tweets* do mesmo tipo/assunto, algo que não é muito esperado dado que existem 25 tipos. Dessa forma, soluções que inseriam *tweets* de outro tipo nesse *cluster*, provavelmente aumentavam muito o custo da solução e eram desconsiderados.

A solução para 10 *clusters* obteve o seguinte número de *tweets* em cada um dos clusters: 27, 20, 8, 21, 29, 18, 27, 64, 28, 9. Da mesma forma que na solução gerada para 5 *clusters*, aqueles com um menor número de *tweets* são os de conteúdo mais homogêneo.

Uma solução ótima para 25 *clusters*, classificaria apenas *tweets* do mesmo tipo em cada um dos *clusters*, porém como é sabido, algoritmos de comportamento estocástico, dificilmente levam a soluções ótimas, ainda assim, consideramos que a solução gerada para os 25 *clusters* obteve boa qualidade, onde após uma inspeção manual, verificamos que dos 251 *tweets* apenas 18 foram classificados inadequadamente.

A Tabela 2 mostra o número máximo de iterações em que o algoritmo não obteve diminuição no custo das soluções. Onde podemos verificar que o número máximo de iterações sem melhoria configurado em 200, possibilitou a geração de indivíduos melhores, em contrapartida, impactou negativamente no desempenho da execução, sendo um *trade-off* que deve ser verificado e ajustado dependendo da aplicação.

# *Clusters*	# Iterações
5	120
10	113
25	144

Tabela 2: Número de iterações sem melhorias nas soluções.

A execução desse trabalho oportunizou uma primeira aproximação entre os participantes com o desenvolvimento de um algoritmo genético para a solução de um problema real mesmo que de cunho experimental e acadêmico. Nesse contexto, características e propriedades pertencentes a este tipo de algoritmo puderam ser verificadas e interpretadas, como por exemplo, as

relações entre tamanho da população x tempo de execução, taxa de mutação x velocidade de conversão, métodos de seleção e cruzamento x variabilidade de soluções, entre tantas outras. Esperamos através de experimentos futuros, fazer melhorias na implementação visando obter resultados mais satisfatórios.