Otimização de Redes Neurais para analise de Sentimento

Raissa Camelo Salhab

[1]Departamento de Computação – Universidade Federal Rural Pernambucana (UFRPE)
Bacharelado em Ciência da Computação

srtacamelo@gmail.com

Resumo. *Os algoritmos evolucionários podem ser grandes aliados para a otimização de pesos de redes neurais artificias. A partir deles pode-se selecionar um conjunto de pesos que favorece a inicialização do treinamento da mesma. As redes neurais por si só são grandes ferramentas ara a analise de sentimento de textos, por tanto, usar um aparato como computação evolutiva para otimiza-las pode faze-las atingirem resultados ainda melhores. O objetivo deste projeto está em utilizar 6 algoritmos evolucionários diferentes para seleção de features, utilizando como base de dados para a rede um conjunto de textos para análise de sentimento.*

1. Introdução

A Mineração de textos se trata da extração de características de documentos escritos, afim de reter algum tipo de informação contido neles. Sendo uma das áreas da ciência de dados, a mineração de textos se preocupa em linkar e classificar conteúdos diferentes e tentar achar padrões para os dados selecionados. [Hearst 2003]

Os computadores são extremamente eficientes em organizar e processar dados numéricos e estruturas de dados computacionais, já que já estão representadas de forma inteligível para o mesmo. Contudo, se tratando de textos escritos em língua humana, as máquinas não são capazes de entender e interpretar com a mesma facilidade que os próprios seres humanos. Se faz necessário algum tipo de mecanismo, do qual textos possam ser processados e informações numéricas e inteligíveis pelo computador possam ser extraídas dos documentos.

Através de técnicas linguísticas como lemmanização (extração de radicais), tokenização (extração de palavras e símbolos), Sentence Spliting (divisão em frases), uso de n-grams e cálculos de frequência de termo a mineração de texto vem possibilitado a interpretação e o processamento de dados textuais por computadores. [Muhr 1991]

Analise de sentimentos é uma das subáreas da mineração de texto, que consiste em classificar se determinado texto contém opiniões positivas, neutras ou negativas, sobre dado contexto. [Leung 2009]

A partir dessa classificação inúmeros tipos de dados podem ser analisados automaticamente, retirando a necessidade de verificar manualmente cada documento um por um. Os processos de analise de sentimentos geralmente envolvem algoritmos de classificação como o KNN ou até mesmo redes neurais, geralmente optando-se por modelos probabilísticos. [Aggarwal and Zhai 2012] Empresas de serviços ao consumidor têm adotado a analise de sentimento como forma de avaliar a aceitação de seus produtos no mercado, servindo como pesquisa de opinião e possibilitando as empresas a adequarem seus produtos de maneira mais eficiente. [Ravi and Ravi 2015] A analise de sentimentos permite não apenas a classificação de de opiniões de usuários sobre produtos como

também tem sido usada como forma de avaliar popularidade de candidatos a cargos políticos em plataformas sociais, avaliação de sentimentos de usuários de redes sociais em relação a determinado assunto ou noticia. [Pozzi et al. 2016][Godbole et al. 2007] Essa área possui um grande potencial a ser explorado, a partir de sistemas de definição de sentimentos, decisões poderão ser tomadas e até mesmo um algoritmo pode determinar que tipo de propaganda exibir para determinado público, quais produtos sugerir, a ideia é cada vez mais a computação torne a experiencia do usuário personalizável e única para cada individuo. Algoritmos evolucionários simulam a evolução biológica natural, criando um esquema de seleção, adaptação, mutação e operações genéticas baseadas na evolução biológica da natureza. Existem vários contextos em que estas aplicações podem ser usadas como heurísticas para problemas NP, maximização/ minimização de funções e otimização de parâmetros para outros algoritmos, como a KNN e a rede neural.
As redes neurais são modelos de aprendizagem de máquina inspirados no sistema nervoso humano. A proposta delas é simular o processo de aprendizagem de um cérebro humano, através de camadas de neurônios artificiais, desta forma fazendo a rede neural artificial reconhecer padrões e objetos. Entre os parâmetros necessários para configurar uma rede neural se encontram a quantidade de camadas, de neurônios, as camadas escondidas e os pesos dos neurônios.
Neste projeto pretende-se utilizar de algoritmos evolucionários para a otimização dos pesos de uma rede neural. Serão usados quatro algoritmos diferentes: Algoritmo genético, estratégias evolucionárias, evolução diferencial e programação evolutiva. Para os quatro algoritmos a função de fitness utilizada será a acurácia da rede treinada e testada com os pesos obtidos pelos mesmos. O objetivo é otimizar uma rede neural para a analise de sentimento de textos.

2. Base de Dados

A base de dados utilizada é a LarissaNet, que consiste em vários textos de reviews (opiniões) sobre produtos das três categorias: jogos (steam), livros (skoob) e músicas (Itunes). Os dados foram obtidos através da utilização de robôs crawlers (Spidering), esses ”robôs”varrem a internet, ou algum site especifico em busca de informações, salvando-as de forma metódica e automatizada.

Os crawlers utilizados para a obtenção desta base de dados foram programados com a linguagem python. Colentando reviews dos produtos citados a cima.
Esse procedimento foi realizado para a obtenção de reviews em português, que são dificilmente encontradas, com esse banco de dados é possível fazer analise de sentimento PT-BR.

Os dados estão organizados em arquivos de textos, dois arquivos para cada categoria (livros, jogos e músicas). Um artigo contendo informações ”positivas”, ou seja, reviews positivos sobre os produtos e outro contendo os reviews ”negativos”. Cada um dos arquivos contém 1.000 textos diferentes, totalizando 2.000 casos por domínio.

Para esse projeto, optou-se por utilizar apenas o dominio de jogos (steam),

usando apenas 500 dados (250 positivos e 250 negativos), afim de diminuir a quantidade de dados processados e também diminuir o tamanho da matriz de dados final. Caso tivéssemos utilizado todos os dados da base a computação dos fitness seria muito custosa, uma vez que a primeira camada da rede neural precisa de uma quantidade de neurônios equivalente a quantidade de colunas da matriz final (após o pré-processamento).

3. Pré-processamento

Primeiramente foi necessário ler cada arquivo txt (steam positivos e steam negativos), extrair cada frase (documento) de cada um destes e atribu-los as suas respectivas classes (0 para negativo e 1 para positivo). Após realizar esta operação as duas listas foram juntadas em uma só (positivas e negativas).

Após ajustada a base de dados alguns procedimentos foram realizados afim de maximizar o potencial de classificação e diminuir o volume dos dados, retendo apenas as informações mais relevantes dos documentos. Através de funções de tokenização e remoção de palavras da biblioteca NLTK, cada documento foi convertido em uma lista de tokens (palavras, sinais, símbolos, siglas). Após a tokenização de cada documento individualmente, a lista tokenizada foi utilizada para remoção de *Stop Words*, palavras repetitivas que não agregam significância à classificação, como pronomes, artigos e preposições. Sinais e números também foram removidos de cada documento. Dada a a remoção de palavras e consequente diminuição do tamanho das listas de tokens gerada por cada documento, o ultimo passo do processamento consistiu em extrair valores numéricos do texto resultante, para poder alimentar os classificadores.

Para facilitar o processo de avaliação da rede neural a base foi aplicada na sua forma numérica. A partir do uso da matrificação TF-IDF, que transforma uma coleção de textos (corpus) em uma matriz númerica, cuja a quantidade de colunas equivale a quantidade de palavras (sem repetições) em todo o corpus. A matriz TF-IDF consiste em um método para computar numericamente as frequências de cada palavra em cada documento, afim de determinar se a frequência de um termo especifico implica na classificação do documento como positivo ou negativo.

Para a extração de valores numéricos dos documentos os conceitos de frequência de termo (TF) e frequência inversa de documento (IDF) foram aplicados. O TF computa a frequência de cada termo em determinado documento, expressado pela formula:

Figura 1. TF

$$TF = nt/n$$

Onde nt é o número de vezes que o termo aparece em um único documento e n é a

quantidade total de termos no documento. O IDF determina a frequência inversa que um termo aparece em todo o Corpus, dado por:

Figura 2. IDF

$$\log(N/dtf)$$

Onde N é a quantidade total de documentos no Corpus e dtf é a quantidade de documentos em que o termo aparece. Os valores de TF e IDF são calculados para todos os termos presentes no corpus. O objetivo do TF é permitir que os classificadores façam relação entre a frequência de determinados termos com a classificação do documento, por exemplo, termos como ”Good”, ”Awesome”e ”Great”tendem a aparecer com mais frequência em documentos de polaridade positiva, enquanto termos como ”Bad”,”Awful”e ”Boring”tendem a aparecer em documentos de polaridade negativa. Porém o uso do TF isoladamente pode fazer com que termos comuns nas duas polaridades assumam valores altos, dado que sua frequência é alta nos dois tipos de documentos, fazendo com que esses termos tenham uma relevância maior para o classificador. Para evitar este problema, o valor do IDF é utilizado como normalizador, pois ao multiplicar o TF com o IDF, documentos que possuem uma frequência muito grande entre documentos distintos tem seu valor diminuído.

Os calculo do TF-IDF foi realizado utilizando as funções da biblioteca sklearn, gerando uma tabela de TF-IDFs onde cada linha da tabela contém os valores de TF-IDF para cada termo em cada documento. A tabela possui um tamanho de d x tn, sendo d a quantidade de documentos no corpus e tn a quantidade total de termos no corpus. A tabela gerada foi salva em formato *.csv*, através da biblioteca pandas, para ser usada posteriormente nos classificadores.

Antes de salvar a tabela tf-idf em formato csv, o valor referente a classificação de cada documento foi adicionado à tabela, (0 para negativo e 1 para positivo). A classificação dos documentos foi conferida através da ordem de leitura dos mesmos antes do pré-processamento. Na base original os arquivos com documentos positivos estão separados em uma pasta e os negativos em outra, apesar da dimensão da base ter sido diminuída, a organização foi mantida, facilitando a caracterização de cada documento. Os primeiros 550 documentos lidos eram positivos, enquanto os 550 seguintes eram negativos, assim só foi necessária a indexação dos valores (0 e 1) nesta ordem, após a concatenação das duas listas de documento e computação dos valores tf-idf.

O passo de calcular os valores tf-idf de cada termo/documento somente após a concatenação das listas de documentos positivos e negativos foi extremamente crucial para a integridade dos dados. O IDF, como visto acima, depende da quantidade total de documentos e de termos do corpus. Caso o tf-idf tivesse sido calculado separadamente para para cada conjunto de dados, os valores seriam significantemente diferentes do que

para o conjunto total. Sem contar que a tabela final de valores seria inutilizável para classificação, uma vez que dificilmente a quantidade de termos seria igual para os dois polos de arquivos, gerando uma matriz não quadrada. E, mesmo que por coincidência a matriz resultante fosse quadrada, os dados não seriam corretos, já que para cada polo de arquivos termos diferentes foram levados em consideração, e sua frequência tf-idf estaria enviesada, dado que a quantidade de documentos utilizada para calcular o idf não iria condiz com a real quantidade total de documentos no corpus.

Como o dataset LarissaNet é muito extenso, o processamento e a classificação dos dados tomariam muito tempo para serem concluídos. Para evitar uma longa espera, dos 6.000 documentos de cada polaridade apenas 250 de cada foram selecionados para o pré-processamento e posterior uso. Ao utilizar a base inteira na primeira vez que o processo foi realizado, os primeiros experimentos levaram um tempo não viável para obter os primeiros valores. Assim, seria inviável continuar com a base inteira.

4. Metodologia

Para a avaliação dos resultados o método de avaliação 70 - 30 % foi utilizado sobre os dados, utilizando 70% dos dados para treinamento e 30% para testes. A tabela gerada durante o pre-processamento foi carregada sobre a forma de um dataframe, da biblioteca pandas, para melhor organizar os dados e fornecer os valores de maneira adequada para os classificadores. Após lidos os valores foram randomizados, usando uma função da própria biblioteca pandas que mistura as linhas dos dataframes sem que os valores das colunas se misturem.
Após randomizados, os dados foram divididos entre frequências e classes, dividindo-os em dois dataframes separados. Para a arquitetura da rede foram escolhidas duas camadas,uma camada de entrada contendo 12 neurônios e a dimensão dos dados (2005) e uma camada de saida com 8 neurônios.

Para a representação dos pesos foi utiliado um vetor numpy de tamanho 12x 2005, ou sejá 12 um array com 12 sub-arrays de 2005 números reais no intervalo entre -1 e 1. Cada cromossomo dos algoritmos evolucionários correspondeu a um conjunto de pesos (12 x 2005). A população utilizada para avaliar os 4 algoritmos foi a mesma, constituindo de 20 cromossomos.
Ao todo foram realizados 50 experimentos para cada algoritmo, inicializando 50 vezes uma única população para os 4 algoritmos, computando 50 vezes a acurácia de cada um dos quatro. A média, a mediana e o desvio padrão de cada conjunto de acurácia foi extraída e a hipótese de t-studant foi utilizada para avaliar as diferenças entre abordagens modificadas dos mesmos classificadores.

5. Modelos Apresentados

5.1. Algoritmo Genético

Os algoritmos genéticos são um tipo de modelo da computação evolutiva que busca encontrar o melhor ”indivíduo”para determinado problema, utilizando de artifícios semelhantes à genética natural. São criados levando como base a teoria da evolução de Darvin, onde o individuo ”mais forte”ou com melhor adaptação ao meio sobrevive e gera descendentes e estes descendentes passam pelo mesmo processo e assim por diante.

O algoritmo funciona inicializando-se primeiro a população e avaliando todos os indivíduos dela, selecionando assim os dois melhores para ”acasalar”. No ”cruzamento”, chamado de crossover os ”genes”da mãe e do pai são misturados, gerando um novo individuo. Após o crossover existe a mutação, que modifica aleatoriamente um dos genes. Esse procedimento pode ser repetido para a mesma população várias vezes, pela decisão do implementador, contanto que não repita os pais e respeite as regras de crossover e mutação.
A mutação e o crossover em si são adaptáveis de acordo com a modelagem do problema e o funcionamento de cada individuo (cromossomo).
Após gerar vários descendentes, estes são misturados com os pais e formam uma nova população para a próxima geração. O algoritmo repete esse procedimento para n gerações.

5.1.1. Modelagem

Para nossa abordagem cada cromosso consistiu em um array de pesos de rede neural, sendo ao todo 20 desses arrays (indivíduos). A função de fitness para avaliar cada conjunto de pesos consistiu em chamar a rede neural passando o peso como parâmetro e avaliando a acurácia do teste.
O número de gerações escolhida para o GA foi 5, gerando a cada geração 10 filhos e mesclando-os com 10 pais aleatórios para formar a população da geração seguinte. Segue abaixo o código da função utilizada.

Figura 3. Código: Algoritmo Genético

```
new_population = []
parents_mated = []
num_generations = 5
for i in range(num_generations):
    fitness_all = ft.calculate_pop_ft(population,x_train, y_train, x_test, y_test)
    for j in range(int((len(population)/2))):
        parents = mating_pool(population,fitness_all,2)
        offspring = crossover(parents,3)
        offspring = mutation(offspring)

        new_population.append(offspring)

        parents_mated.append(parents[0])
        parents_mated.append(parents[1])

    new_population = np.concatenate((new_population,parents_mated[:int((len(population)/2))]))
    population = new_population

    #------- Clean aux Lists------
    parents_mated = []
    new_population = []

best = ft.find_best(population,x_train, y_train, x_test, y_test)
```

5.2. Estratégia Evolutiva

Na ciência da computação, estratégia evolutiva é uma técnica de optimização baseada nas ideias de adaptação e evolução. Também inspirado na teoria da evolução das espécies e na perpetuação dos indivíduos mais fortes.

5.2.1. Modelagem

As estratégias evolucionárias funcionam de forma parecida ao algoritmo genético, porém levando em consideração maior os fatores aleatórios. Nela a população não precisa ser avaliada inicialmente, apenas escolhendo aleatoriamente os pais do novo individuo a ser gerado. Após escolher os pais realiza-se o crossover e a mutação iguais ao ga.

Neste projeto esse procedimento foi realizado 25 vezes para cada geração, gerando 25 indivíduos novos. Dentro destes 25 são escolhidos os 20 melhores, avaliando o fitness de cada um. Estes 20 melhores constituem a nova população para a próxima geração. O número de gerações escolhido para esse algoritmo foi 3.
A seguir o código implementado.

Figura 4. Código: Estratégia Evolutiva

```
num_generations = 3
decendants = []
for i in range(num_generations):
    number_decendants = 0

    while number_decendants < (len(population)+5):
        #print(population)
        father = np.random.random_integers(0,len(population)-1)
        mother = np.random.random_integers(0,len(population)-1)

        father = population[father]
        mother = population[mother]


        crossed = crossover(father,mother,1)
        mutated = mutation(crossed)
        #print(crossed)
        decendants.append(mutated)

        number_decendants += 1

    ft_decendants = ft.calculate_pop_ft(decendants, x_train, y_train, x_test, y_test)
    new_population = fetch_n_better(decendants,ft_decendants,len(population))
    population = new_population
ee_accu = ft.fitness(population[0],x_train, y_train, x_test, y_test)
```

5.3. Evolução Diferencial

A evolução diferencial é um algoritmo usado especialmente para otimização de algoritmos caixa preta, como redes neurais. Consiste em técnicas evolutivas comuns aos algoritmos genéticos e estratégias evolutivas, porém a geração de novos individuos se da de uma maneira diferente. Na evolução diferencial para cada individuo na população são escolhidos outros 3 indivíduos diferentes deste, chamaremos de a,b e c. Então é feita a subtração dos genes de a por b seguida da soma com os genes de c.
Esse algoritmo é usado principalmente quando os cromossomos do problema são modelados como arrays numéricos, e cada gene é um valor.

5.3.1. Modelagem

Na implementação feita, iterou-se sobre todos os indivíduos da população, gerando um novo individuo usando o processo descrito a cima. Depois de gerar o novo individuo são selecionadas aleatoriamente quais ”partes”(genes) do novo individuo iram substituir os genes do membro da população selecionado. Então o fitness do novo individuo criado é comparado com o antigo, caso seja maior, substitui o mesmo na população.
A seguir o código implementado.

Figura 5. Código: Evolução Diferencial

```
fitness_all = ft.calculate_pop_ft(population, x_train, y_train, x_test, y_test)
weight_dimention = (2005,2)
mut_constant = 0.5
limiar_crossover = 0.6
best_idx = 0
for i in range(len(population)):  #Iterate over all cromossomes
    indices = [indices for indices in range(len(population)) if indices != i]
    a, b, c = population[np.random.choice(indices, 3, replace=False)]
    mutant = np.clip(a + mut_constant * (b - c), -1, 1)
    cross_points = np.random.uniform(low=0, high=1,size=weight_dimention) < limiar_crossover
    trial = np.where(cross_points, mutant, population[i])
    trial_ft = ft.fitness(trial,x_train, y_train, x_test, y_test)
    if trial_ft > fitness_all[i]:
        fitness_all[i] = trial_ft
        population[i] = trial_ft
        if trial_ft > fitness_all[best_idx]:
            best_idx = i
            #best_weight = trial
return fitness_all[best_idx]
```

5.4. Programação Evolutiva

Também inspirado no processo de seleção natural da evolução a programação genética foca mais nos aspectos hereditários relacionados ao fenótipo, sem necessariamente realizar a operação de crossover entre dois indivíduos.

5.4.1. Modelagem

A implementação feita funciona primeiramente avaliando todos os membros da população,logo em seguida salvando o melhor destes. Então para cada membro da população gera-se um novo individuo apartir da mutação (mesma mutação dos outros algoritmos). Após gerar as mutações (20 mutações de 20 indivíduos da população original), calcula-se o fitness dos novos indivíduos e novamente, salvando o melhor entre estes.
Em seguida junta-se a lista com a população inicial com a lista dos novos indivíduos, gerando uma lista de tamanho 40. Então para cada cromossomo (Si) da lista são escolhidos 3 cromossomos aleatórios também da mesma lista. Estes três cromossomos tem o fitness comparado ao fitness de Si. Caso Si tenha o maior fitness do que pelo menos 2 dos3 cromossos aleatórios, o cromosso Si é adicionado a lista dos melhores individuos, caso contrário o melhor dos 3 cromossomos aleatórios é adicionado.
Ao final deste procedimento extrai-se os 20 melhores cromossomos do array do smelhores, formando a nova população para a próxima geração. São 3 gerações.

Figura 6. Código: Programação Evolutiva

```
    for parent in population:
        child = mutation(parent)
        children.append(child)

    children_ft = ft.calculate_pop_ft(children, x_train, y_train, x_test, y_test)
    max_fitness_idx = np.where(all_fitness == np.max(all_fitness))
    max_fitness_idx = max_fitness_idx[0][0]

    best_child = population[max_fitness_idx]
    best_child_ft = all_fitness[max_fitness_idx]

    best_solution, best_solution_ft = best(best_child,best_child_ft,best_solution,best_solution_ft)

    union = np.concatenate((population,children))
    union = union.tolist()
    fitness = np.concatenate((all_fitness,children_ft))
    fitness = fitness.tolist()
    wins_idx = []
    for si in union:
        idx_si = union.index(si)
        for l in range(boutsize):
            sj = random.choice(union)
            idx_sj = union.index(sj)
            if(fitness[idx_si] > fitness[idx_sj]):
                si_wins += 1
            else:
                actual_idx_sj = idx_sj
        if si_wins > 1:
            wins_idx.append(idx_si)
        else:
            wins_idx.append(actual_idx_sj)
    population = selectBest(union,len(population),wins_idx)
return best_solution_ft
```

6. Resultados

A seguir encontra-se os resultados obtidos através dos 50 experimentos realizados em cada algoritmo. Primeiro, segue a tabela com os valores de média, desvio padrão e mediana.

Figura 7. Resultados

Média, Desvio Padrão e Mediana do Tempo de Execução			
Algoritmo	Média	Desvio Padrão	Mediana
Algorítmo Genético	0.642	0.022	0.65
Estratégias Evolucionárias	0.706	0.114	0.75
Evolução Diferencial	0.735	0.053	0.73
Programação Evolutiva	0.622	0.081	0.64
Estratégias Evolucionárias - C	0.642	0.034	0.65
Estratégias Evolucionárias -CC	0.6427	0.034	0.65
Random Forrest	0.6154	0.094	0.65
GSO	0.746	0.056	0.77
PSO	0.709	0.064	0.69

Como pode-se obvervar, os reusultados não variam muito de um algoritmo pro outro, obtendo em média uma acurácia de 70%. O desvio padrão também é baixo, o que indica que a distribuição do conjunto de acurácias também é pequena.
Ou seja, tecnicamente todos os quatro algoritmos obtiveram a mesma faixa de resultado,

não houve grandes alterações na média geral.

Para entender mais a fundo o impacto que cada técnica teve iremos analisar também os valores obtidos com a hipótese de t-studant, comparando cada uma das técnicas umas com as outras.

A seguir encontra-se o resultado das hipóteses de estática implementada, usando t-studant.

Figura 8. Resultados

Comparação		
Comparação	t-value	p-value
Normal e GA	-0.899	0.3790
Normal e EP	-0.193	0.848
Normal e EE	-2.0340	0.0554
Normal e DE	-3.667	0.0015
Normal e EE_Cauchy	-0.899	0.379
Normal e EE_Cauchy2	-0.899	0.3790
Normal e GSO	-4.609	9.414
Normal e PSO	-2.625	0.0166
GA e EP	0.746	0.4642
GA e EE	-1.764	0.8848
GA e DE	-4.797	0.6047
GA e ee_cauchy	0.0	1.0
GA e ee_cauchy2	0.0	1.0
GA e GSO	0.7461	0.4642
GA e PSO	-1.764	0.0929
EP e EE	0.1510	0.8848

Figura 9. Resultados

Comparação		
Comparação	t-value	p-value
EP e DE	-0.5460	0.6047
EP e ee_cauchy	-0.746	0.464
EP e GSO	-47.427	6.6278
EP e PSO	-2.664	0.0153
EE e ee_cauchy2	1.764	0.092
EE e DE	-0.6632	0.5318
EE e GSO	-1.238	0.226
EE e PSO	-0.0641	0.9495
DE e ee_cauchy	4.7976	0.0001
DE e ee_cauchy2	4.7976	0.0001
DE e GSO	0.512	0.612
DE e PSO	1.0244	0.3184
GSO e PSO	1.576	0.1273

Em todos os casos o p-value foi maior que o t-value, exceto para os algoritmos GA e EP, o que indica que há uma probabilidade muito baixa destes outros casos de que haja alguma diferença entre os conjuntos de acurácia.

7. Considerações Finais

Como foi visto os algoritmos não tiveram grandes diferenças entre si, obtendo resultados semelhantes. Para próximas etapas do projeto pode-se testar utilizar o erro da rede neural como valor de fitness para os algoritmos, mudar a arquitetura da rede, adicionando mais camadas e testando mudar a quantidade de neurônios.

Outro fator impactante no projeto foi o tempo gasto nos experimentos, chegando a vários dias para completar os 50 experimentos de um só algoritmo. Isso pode ser trabalhado nas próximas versões do projeto, mudando a implementação dos algoritmos para otimiza-los, diminuindo o tamanho da população ou a quantidade de pesos por camada (mexer na arquitetura da rede). Em futuros trabalhos podemos também pré-processar melhor os dados para diminuir o volume de colunas na matrix de TF-IDF, desta forma também diminuindo a quantidade de neurônios para a camada inicial, consequentemente a quantidade de pesos.

Alterações e revisões deverão ser feitas para as próximas etapas do projeto, visando também realizar alterações onde for necessário e encontrar possíveis erros. Novos algoritmos podem também serem testados.

Referências

Aggarwal, C. C. and Zhai, C. (2012). *Mining text data*. Springer Science & Business Media.

Godbole, N., Srinivasaiah, M., and Skiena, S. (2007). Large-scale sentiment analysis for news and blogs. *Icwsm*, 7(21):219–222.

Hearst, M. (2003). What is text mining. *SIMS, UC Berkeley*.

Leung, C. W. (2009). Sentiment analysis of product reviews. In *Encyclopedia of Data Warehousing and Mining, Second Edition*, pages 1794–1799. IGI Global.

Muhr, T. (1991). Atlas/ti—a prototype for the support of text interpretation. *Qualitative sociology*, 14(4):349–371.

Pozzi, F. A., Fersini, E., Messina, E., and Liu, B. (2016). *Sentiment analysis in social networks*. Morgan Kaufmann.

Ravi, K. and Ravi, V. (2015). A survey on opinion mining and sentiment analysis: tasks, approaches and applications. *Knowledge-Based Systems*, 89:14–46.