André S. de Almada 9292952

Leonardo Alves Gomes 9293178

Classificação textual através da representação de Bag-of-Words e do algoritmo Naïve Bayes

São Carlos

2018

1. Introdução

A mineração de textos é uma complicada tarefa na computação que consiste em classificar conjuntos de textos de acordo com o seu conteúdo, através da busca por padrões. O objetivo deste documento é apresentar ao leitor uma estratégia de classificação textual em 574 artigos científicos, divididos em três grandes temas: raciocínio baseado em casos (CBR), programação lógica indutiva (ILP), e recuperação de informação (IR).

Para alcançar a meta definida anteriormente, é necessário realizar uma etapa de pré-processamento nos conjuntos de textos (*corpus*), escolher alguma forma de representação para eles e, por fim, adotar um algoritmo para classificação e treinamento do modelo escolhido. A proposta deste projeto é a utilização do modelo *bag-of-words* para representação dos textos e o uso do algoritmo de *Naïve-Bayes* para o treinamento e classificação dos dados.

2. Considerações Iniciais

Para a execução do projeto foi escolhida a utilização de uma popular distribuição de ciência de dados para Python, chamada **Anaconda**, em sua versão 5.3, lançada em setembro deste ano. Esta distribuição, livre e de código aberto, facilita a instalação de mais de 1400 pacotes de ciência de dados para as linguagens de programação Python e R, gerenciando suas dependências e ambientes.

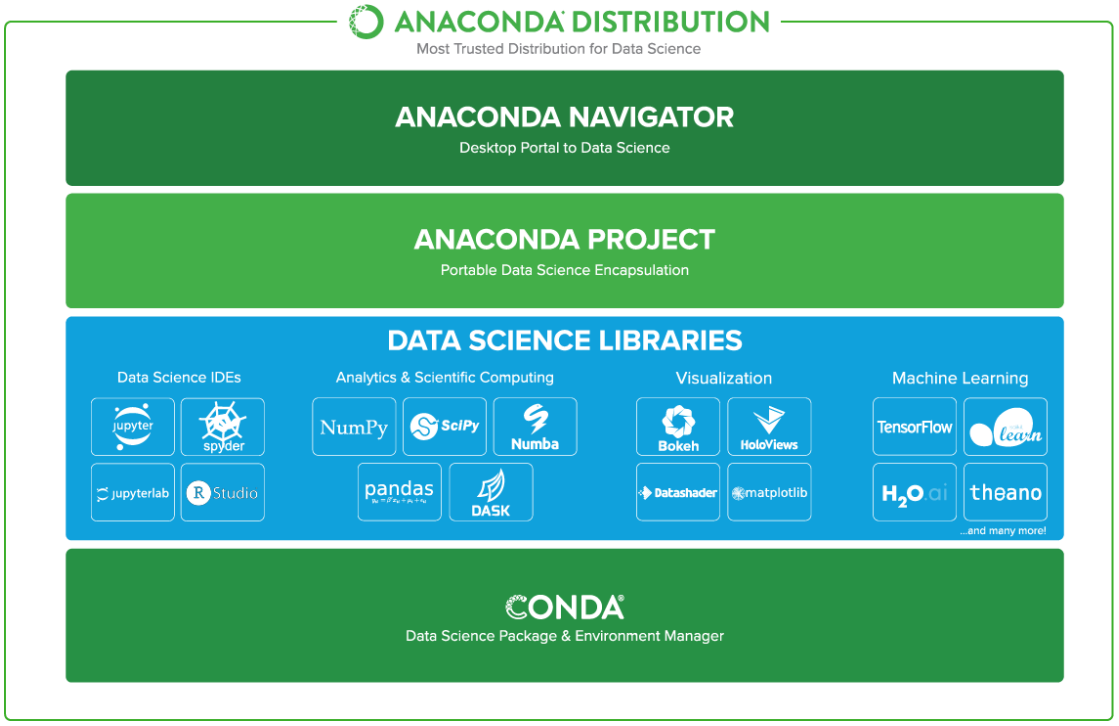

Figura 1. Visualização da distribuição Anaconda

Com o Anaconda instalado, a utilização dos conjuntos de bibliotecas **scikit-learn**, **Pandas** e **NLTK** tornou-se possível. O primeiro foi empregado para que fosse possível a representação do *corpus* pelo modelo *bag-of-words* e para dividi-lo em 10 folds e realizar o procedimento de validação cruzada 10-folds. O segundo, por sua vez, foi utilizado para obtermos uma matriz *"tema - documento"*, organizando melhor os dados fornecidos entre classe e texto, além de poder exportá-la mais tarde no modelo CSV. Por fim, a biblioteca NLTK, abreviação de Natural Language Toolkit, é utilizada no pré-processamento dos textos, para os processos de tokenização, retirada de stopwords e stemização.

Com a tabela exportada para CSV, também foi possível a utilização do Weka, um pacote de softwares para aprendizado de máquina, com o propósito de comparar a acurácia do algoritmo desenvolvido neste projeto com outros.

3. Pré-processamento

Para a etapa de pré-processamento, cada arquivo é aberto e tem do seu nome extraído o tema que aborda. Após a identificação do tema, são retirados os caracteres não alfanuméricos e o texto todo é colocado na minúscula, para que não haja diferença entre *"Academia"* e *"academia",* por exemplo.

Com o corpus neste formato, foram utilizadas as funções providas pelo NLTK para as técnicas de tokenização, stemização e retirada de stopwords.

```
# caminho da pasta com os artigos
articles_dir = './txt/'
# cria uma lista com os caminhos dos arquivos de artigos
filenames = [f for f in glob.glob(os.path.join(articles_dir,
'*.txt'))]

stemmer = PorterStemmer()
# Lê o texto de um arquivo, fazendo os devidos pré-processamentos
def get_data (filename):
    with open(filename, 'r', encoding='latin1') as f:
        corpus = f.read() # leitura do corpuso do arquivo
        corpus = re.sub('[^A-Za-z]', ' ', corpus) # Retira
caracteres não alfanumericos
        corpus = corpus.lower() # Torna todas as palavras
minúsculas
```

```
        corpus = word_tokenize(corpus) # tokenização do corpuso
        # remoção das stopwords
        for token in corpus:
            if token in stopwords.words('english'):
                corpus.remove(token)

        for i in range(len(corpus)): # Processo de Stemming
            corpus[i] = stemmer.stem(corpus[i])

        plain_corpus = " ".join(corpus) # Transforma o array de
tokens em uma string única por artigo
        return plain_corpus

# Pega o tema do artigo através do nome do arquivo
def get_class (filename):
    return filename.split('-')[0].split('\\')[1]

print("lendo arquivos e executando pré-processamento...")
# criação de um dataframe do pandas com uma coluna de temas e de
textos
dataset = pd.DataFrame({'theme': [get_class(f) for f in
filenames],
                        'data': [get_data(f) for f in filenames]})
```

Alguns artigos ainda mencionam a adição de um método para corrigir a ortografia do corpus, entretanto como os dados são artigos científicos, é assumido que a ortografia está bem escrita e revisada, permitindo não implementar esta etapa.

4. Bag-Of-Words

Para a criação do modelo *bag-of-words*, o scikit-learn provê uma função chamada CountVectorizer, que contabiliza a frequência das palavras nos textos. Como parâmetro, escolhemos apenas armazenar os 1000 elementos mais frequentes.

```
print("transformando textos em matriz termo-documento...")
# transformação em matriz de termo/documento
cv = CountVectorizer(max_features=1000, encoding='latin1')
```

```
X = cv.fit_transform(dataset.data).toarray() # Bag of words de
cada artigo
words = cv.get_feature_names() # palavras presentes em X
y = dataset.theme # Classificação dos artigos

# matriz de termo/documento com coluna de classe
tdm = pd.concat([y, pd.DataFrame(X, columns=words)], axis=1)
```

No final da execução, é criada uma matriz onde cada coluna é uma das mil palavras que aparecem com mais frequência nos textos, com exceção da primeira que indica a classe do texto, e cada linha representa um artigo disponibilizado no corpus. Assim, uma cédula representa a frequência do termo no texto.

5. Naïve-Bayes

A implementação segue o algoritmo ensinado em sala de aula.

```
# determina a classe de um documento utilizando naive bayes
def bayes(document, train_set, themes):
    # dicionário que guarda as probabilidades de cada tema
    probabilities = {}

    # calcula a probabilidade condicional (parcial) de cada tema
    for theme in themes:
        # separa a porção do dataset associada ao tema atual
        theme_set = train_set[train_set.theme == theme]
        # calcula a probabilidade do tema atual (P(A))
        theme_prob = (len(theme_set) / len(train_set))

        # calcula a probabilidade condicional para cada palavra
(P(B|A))
        for word in theme_set.columns[1:]:
            theme_prob *= ((len(theme_set[theme_set[word] ==
document[word]]) + 0.1) /
                           (len(theme_set) + 0.1))

        # guarda o valor no dicionário de probabilidades
        probabilities[theme] = theme_prob
```

```
    # retorna a entrada no dicionário
    return max(probabilities, key=lambda k: probabilities[k])
```

Uma observação interessante de ser feita é de que, neste caso, não é necessário realizar o cálculo da probabilidade do documento ser ele mesmo no conjunto de textos, pois será o mesmo valor para o cálculo das três classificações possíveis (CBR, ILP e IR), tornando-a irrelevante para a escolha da classe.

Para evitar que as probabilidades com valor 0 prejudicassem o aprendizado do classificador, foi utilizado o estimador de Laplace, somando 0.1 no numerador e no denominador do cálculo da probabilidade.

6. Etapa de testes

Utilizando a função StratifiedKFold do scikit-learn, definimos 10 folds para utilizar nos testes, embaralhando-os entre um teste e outro.

```
# gerador de 'folds' normalizadas para um k-fold cross-validation
n_splits = 10 # quantidade de folds
skf = StratifiedKFold(n_splits=n_splits, shuffle=True)

print(f"realizando {n_splits}-fold cross-validation...")

# inicialização da acurácia
accuracy = 0
# criação do conjunto de temas possíveis
themes = list(set(tdm.theme))
count = 0 # contagem de folds
for train_index, test_index in skf.split(X, y):
    # separação entre teste e treinamento
    train_set = tdm.iloc[train_index]
    test_set = tdm.iloc[test_index]

    print(f"fold {count} ({len(test_set)} itens):")

    correct_count = 0 # contagem de exemplos corretamente
previstos
    for index, doc in test_set.iterrows():
        # calcula a predição para um exemplo do conjunto de tetes
```

```
        pred = bayes(doc, train_set, themes)
        # incrementa contagem de previsões corretas caso esta seja
correta
        correct = (pred == doc.theme)
        correct_count += 1 if correct else 0
        print(f"{'.' if correct else 'x'}", end='', flush=True)

    correct_count /= len(test_set)
    print(f"\nfold {count} accuracy: {correct_count}")

    # adiciona à acurácia total a acurácia obtida neste fold
    accuracy += correct_count / len(test_set)

    # incrementa a contagem de folds
    count += 1

# divide a acurácia pela quantidade de folds, efetivamente
calculando a média
# das acurácias
accuracy /= n_splits

print (accuracy)
```

7. Resultados

A saída do programa exibe a execução dos 10 testes, sendo que o caractere '.' indica um acerto na predição, enquanto 'x' um erro.

```
realizando 10-fold cross-validation...
fold 0 (58 itens):
..........................................................
fold 0 accuracy: 1.0
fold 1 (58 itens):
..x.........................x.....................x.x....
fold 1 accuracy: 0.9310344827586207
fold 2 (58 itens):
.................x..x.....................x....x..........
fold 2 accuracy: 0.9310344827586207
fold 3 (58 itens):
.....................................x.x.........x...
```

```
fold 3 accuracy: 0.9482758620689655
fold 4 (58 itens):
.x.................x................................x....
fold 4 accuracy: 0.9482758620689655
fold 5 (58 itens):
.........................x............................x.x.
fold 5 accuracy: 0.9482758620689655
fold 6 (57 itens):
............x.........................x.................
fold 6 accuracy: 0.9649122807017544
fold 7 (57 itens):
.x.....................................................
fold 7 accuracy: 0.9824561403508771
fold 8 (57 itens):
.....................................xx...............
fold 8 accuracy: 0.9649122807017544
fold 9 (55 itens):
.....................................x..x.....x.....x.
fold 9 accuracy: 0.9272727272727272
0.95464499807
```

8. Comparações com outros algoritmos

Para verificar a eficiência do algoritmo implementado neste projeto, foi realizado testes com diferentes algoritmos de classificação fornecidos no WEKA. Assim, foi exportada a matriz fornecida pela biblioteca Pandas para um arquivo csv, com as etapas de pré-processamento já feitas.

Com o CSV em mãos, o WEKA foi empregado para gerar um arquivo .arrf no qual foram aplicados dois filtros: *NominalToString*, convertendo o atributo que era do tipo "nominal" para "string", possibilitando o uso do segundo filtro, *StringToWordVector*, responsável por fazer a representação no modelo *bag-of-words*, também selecionando os 1000 termos mais frequentes.

Com o ambiente já definido, foram escolhidos quatro algoritmos: a árvore de decisão **J48** , **KNN**, **SMO** (Sequential Minimal Optimization, uma resolução do classificador SVM que é mais eficiente que os classificadores comuns) e o próprio **Naïve-Bayes**.

Naïve-Bayes	J48 (Weka)	KNN (Weka)	SMO (Weka)	Naïve-Bayes (Weka)
95.4645 %	94.7735 %	92.6829 %	99.3031 %	98.4321 %

Tabela 1. Comparação da acurácia entre diferentes classificadores

Referência Bibliográficas

ANACONDA. **Anaconda Distribution**: The Most Popular Python/R Data Science Distribution. Disponível em: <https://www.anaconda.com/distribution/>. Acesso em: 24 out. 2018.

RAJENDRAN, C. **Text classification using the Bag Of Words Approach with NLTK and Scikit Learn**. Disponível em: <https://www.linkedin.com/pulse/text-classification-using-bag-words-approach-nltk-scikit-rajendran>. Acesso em 25 out. 2018.

APPALARAJU, S. **SVM and SMO main differences**. Disponível em: <https://stats.stackexchange.com/questions/130293/svm-and-smo-main-differences>. Acesso em 6 nov. 2018.