Relatório Técnico do Trabalho 2 da Disciplina de Inteligência Artificial

Eduardo José Silva, Igor Brehm, Israel Deorce Vieira Jr e Lucas Cerqueira Stein

Escola Politécnica – PUCRS

90619-900 Porto Alegre/RS, Brasil

25 de junho de 2019

Resumo

Este artigo descreve as etapas de desenvolvimento do segundo trabalho proposto na disciplina de Inteligência Artificial do curso de Engenharia de Software da Escola Politécnica da PUC, envolvendo a construção do corpus, seu processamento e classificação e a análise de resultados no contexto de um sistema de perguntas e resposta em Língua Portuguesa.

Objetivo do trabalho

O presente trabalho tinha como objetivo estimular por parte dos alunos o uso das técnicas e das ferramentas estudadas ao longo do semestre, bem como o desenvolvimento das habilidades de análise crítica de seus resultados.

Escolhemos codificar a solução na linguagem Python, e nos valemos, ainda, de duas APIs, sendo uma delas a *openpyxl*, destinada a manipular arquivos xlsx, e a segunda uma aplicação, disponível em repositório aberto no GitHub[1], que fornece uma interface para utilização da ferramenta Cogroo compatível com Python.

Passamos, então, a detalhar os passos que seguimos, apresentando primeiramente, a aplicação que desenvolvemos e, por derradeiro, nossas análises a respeito da saída fornecida pelo Weka.

Construção do corpus

O corpus utilizado no presente trabalho consiste em uma coleção de perguntas e resposta versando sobre o algoritmo Random Forest (Floresta Aleatória). O corpus foi construído coletivamente pelos alunos da disciplina, tendo cada um deles elaborado cerca de 20 perguntas com resposta sobre o algoritmo, envolvendo os temas "conceito", "propósito", "paradigma", "tarefa", "funcionamento", "parâmetros", entre outros correlatos.

Foi estipulado que que para cada pergunta deveriam ser fornecidas ao menos 5 variantes bem definidas, devendo, para tanto, ser utilizadas variações linguísticas, paráfrases e outros recursos análogos. O formato escolhido para o corpus foi uma planilha do Excel (todos.xlsx), contendo as colunas "ID", "PERGUNTA", "RESPOSTA", "AUTORES" e "CLASSES". O referido arquivo foi disponibilizado na plataforma Moodle da disciplina e continha um total de 1203 entradas, sem contar o cabeçalho.

Pré-processamento

Iniciamos o trabalho pela revisão e consolidação das classes de perguntas. Nesta etapa, identificamos e corrigimos diversas ocorrências de dados faltantes.

Com efeito, no corpus original diversas perguntas não contavam com classificação. Visando possibilitar seu aproveitamento, nos casos em que se concluiu se tratar de pergunta válida, contendo enunciado e resposta, estando faltante apenas a

1

classificação, manualmente atribuímos à pergunta a classe que julgamos ser a mais apropriada dentre as já existentes.

Após, tratamos os ruídos nos dados decorrentes de erros de digitação, que resultaram na unificação de várias perguntas em uma só categoria. Exemplificando, ao observarmos a existências das categorias " funcionamento" e "funcionamento ", optamos por eliminar a segunda delas e por manter a primeira.

Também nos deparamos com situações de inconsistências nos dados. Analisamos de forma crítica a classificação atribuída a cada uma das perguntas e optamos por sua reclassificação nas situações em que havia incoerência entre o respectivo conteúdo e a categoria a ela atribuída.

Por fim, procedemos à eliminação dos dados inconsistentes, tais como linhas vazias na planilha, perguntas sem respostas (ou vice-versa) e ausência de classe – considerando, aqui, que já havíamos solucionado, quando possível, as ocorrências de dados faltantes nessa coluna, de forma que restavam então apenas situações de inconsistência insanável.

Concluída esta etapa inicial, foi criado um novo arquivo (corpus.xlsx), contando com o total de 1.008 linhas, sobre o qual foi possível criar um conjunto de classes "sobreviventes", que organizamos em um armazenamos em um set para consulta futura.

Abaixo, apresentamos o método responsável pela execução de tais tarefas:

```
def lerPerguntas():

        for i in range(1, max_row+1):
                enunciado = planilha.cell(row = i, column = 2).value
                classe = planilha.cell(row = i, column = 4).value

                if enunciado != None and classe != None:
                        pergunta = [i, enunciado, classe]
                        perguntas.append(pergunta)

                        if classe not in classes:
                                classes.add(classe)

        arquivo.save(filename = "corpus.xlsx")
```

Normalização das classes e das perguntas

Concluído a primeira parte da etapa de pré-processamento do corpus, procedemos à normalização dos dados das perguntas e das classes remanescentes. A técnica utilizada

para tanto foi a de lematização. Aplicamos sobre os dados a ferramenta Cogroo, que integramos ao código por meio da importação de uma API aberta[1].

A tarefa foi executada por meio de código em Python, utilizando-se do acionamento do método lematize(String) fornecidos pela API openpyxl, cujo retorno é uma string, sobre cada uma das perguntas:

```
def lematizar(texto):
        return cogroo.lemmatize(texto)
```

A figura abaixo ilustrar a execução do método:

```
Texto original:
O pescoço da girafa é cheio de pintinhas e vai do chão até o céu
Texto lematizado:
o pescoço de o girafa ser cheio de pintinhas e ir de o chão até o céu
```

Análise morfológica

Na sequência, a ferramenta Cogroo foi utilizada para proceder à análise morfológica das perguntas, visando a extração de tokens.

A tarefa foi executada por meio de código em Python, utilizando-se do método *analyze(String)*, fornecido pela API openpyxl:

```
def analisar(texto):
        return cogroo.analyze(texto)

def extrairTokens(texto):
        return texto.sentences[0].tokens
```

Como retorno, obtêm-se uma lista de tokens, conforme exemplo:

```
Tokens:
[o#art M=S, pescoço#n M=S, de#prp -, o#art M=S, girafa#n F=S, ser#v-inf -, cheio
#adj M=S, de#prp -, pintinhas#n F=P, e#conj-c -, ir#v-inf -, de#prp -, o#art M=S
, chão#n M=S, até#prp -, o#art M=S, céu#n M=S]
```

Criação da Bag of Words - BoW

Para criar a Bag of Words, iniciamos construindo, para cada classe de perguntas, um dicionário dos termos mais recorrentes, ao qual associamos o número de vezes em que

[1] Disponível em

aquele token foi utilizado nas perguntas da classe. Considerando as características do corpus, selecionamos como classes de tokens mais relevantes apenas os advérbios (“adv”), os verbos no infitinivo (“v-inf”) e os nomes próprios (“prop”).

Assim, iterando sobre as perguntas de uma determinada classe, procedemos à leitura de seus tokens, um a um, a fim de verificar, em primeiro lugar se o token já havia sido incluído no dicionário.

Caso negativo, ele era incluído como chave no dicionário e associado a ele o valor “1”, indicando ser a primeira vez que ele era encontrado dentro daquela classe de perguntas; caso o token fosse encontrado como chave no dicionário, bastava incrementar em uma unidade o valor a ele associado, entendendo-se que ele já havia sido ocorrido antes:

```
def criarBoW():
        for classe in dictDePerguntas: #pego todas as perguntas de cada classe
                tokens = dict() #gero um dicionario dos tokens
                lista = dictDePerguntas[classe]
                for i in range (len(lista)):
                        pergunta = lista[i]
                        pergunta = pergunta[4] #pego os tokens da pergunta atual
                        for token in pergunta:
                                if token not in tokens.keys():
                                        tokens[token] = 1
                                else:
                                        tokens[token] = tokens[token] + 1
```

Uma vez formada a lista de tokens mais recorrentes dentro da classe, delimitamos que os cinco termos de maior frequência deviam ser acrescidos à lista geral de tokens que serviria de substrato à formação da Bag of Words. Destacamos que tomamos o cuidado de não reincluir na Bag of Words termos em duplicidade.

Assim, ainda dentro do método criarBow(), executamos o seguinte trecho de código:

```
sorted_tokens = sorted(tokens.items(), key=operator.itemgetter(1),reverse=True)
for j in range(5):
        aux = sorted_tokens[j][0]
        if(aux not in listaGeral):
                listaGeral.append(aux)
```

A lista de termos selecionados para integrar a Bag of Words ficou assim definida:

afetar	ir
aprender	lidar
aumentar	limitar

Bagging	max
citar	modificar
como	oob
comparar	possuir
criar	quando
dar	quão
Decision_Tree	Random_Forest
diferir	Random_Forests
elevar	ser
evitar	servir
existir	significar
Floresta	supervisionar
Floresta_Aleatória	tender
Florestas	ter
forest	treinar
formar	usar
funcionar	utilizar
implementar	variar

Representando as perguntas

A última etapa envolvendo o pré-processamento do corpus foi a estruturação das perguntas. Optamos por representar cada pergunta como um vetor binário de igual número de posições da quantidade de tokens existentes na lista que compõe Bag of Words, de forma que, considerando a lista ordenada, o número 0 indica que a pergunta não contém aquele token, enquanto que o número 1 sinaliza que o token está presente na pergunta em análise.

O método escrito para a tarefa foi o seguinte:

```
def estruturar():
        for i in range(len(perguntas)):
                vector = list()
                aux = perguntas[i]
                for j in range(len(aux[4])):
                        palavra = aux[4][j]
                        for k in range(len(listaGeral)):
                                if(palavra == listaGeral[k]):
                                        vector.append(k)
                result = list()
                for w in range(len(listaGeral)):
                        result.append(0)
                for index in vector:
                        result[index] = 1
                perguntas[i].append(result)
```

Processamento pelo Weka

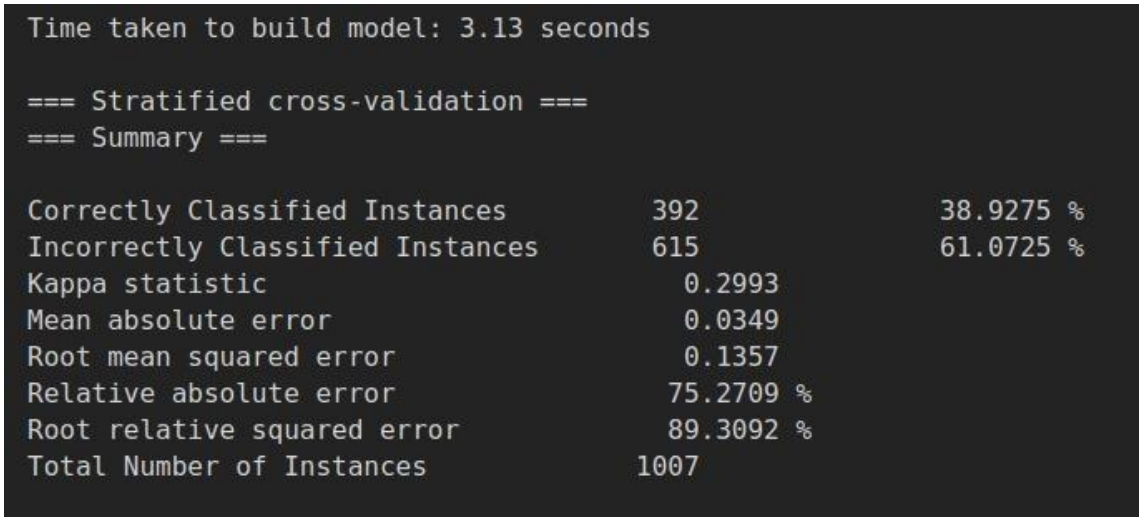
Em primeiro lugar foi carregado no Weka o arquivo no formato *.arff* que havia sido gerado pelo código Python abaixo:

```
def criarOutput():
        f= open(outpath,"w+")
        f.write("@relation Arquivo\n\n")
        for i in range(len(listaGeral)):
                f.write("@attribute P"+str(i) +" numeric"+"\n")
        aux = str(classes)
        aux = aux.strip('[]')
        f.write("@attribute classe "+aux+"\n")
        f.write("\n@data\n")
        for i in range(len(perguntas)):
                aux = str(perguntas[i][5])
                aux = aux.strip('[]')
                f.write(aux+", "+perguntas[i][2] +"\n")
        f.close()
```

Na sequência, os modelos foram treinados utilizando validação cruzada, com cinco folds cada. Os algoritmos empregados foram o MultiLayer Perceptron, o Random Forest e o BaeysNet.

Análise dos resultados

Considerando não apenas o índice de acertos, mas também o tempo de execução, concluímos que o algoritmo que apresentou os melhores resultados foi o Random Forest, ficando em segundo lugar o Multilayer Perceptron e, por último, BayesNet.



Random Forest

```
Time taken to build model: 75.41 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         400               39.7219 %
Incorrectly Classified Instances       607               60.2781 %
Kappa statistic                          0.308
Mean absolute error                      0.0348
Root mean squared error                  0.1373
Relative absolute error                 74.9584 %
Root relative squared error             90.3245 %
Total Number of Instances             1007
```

Multilayer Perceptron

```
Time taken to build model: 0.55 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         246               24.429  %
Incorrectly Classified Instances       761               75.571  %
Kappa statistic                          0.0948
Mean absolute error                      0.0441
Root mean squared error                  0.148
Relative absolute error                 95.2025 %
Root relative squared error             97.3527 %
Total Number of Instances             1007
```

BayesNet

Abaixo, sumarizados os dados de *true positives* (TP), *false negatives* (FB) e de *recall* extraídos da matrix de confusão do algoritmo do MultiLayer Perceptron:

Classe	**TP**	**TP + FN**	**Recall**
a = caracteristica	0	5	0%
b = tratamentoDeDados	7	10	70%
c = maxFeatures	0	15	0%
d = pre-processamento	2	5	40%
e = conceito	41	121	34%
f = metodo	0	10	0%
g = parametro	12	15	80%
h = pos-processamento	5	5	100%
i = paradigma	4	20	20%

j = diferenca	2	5	40%
k = hiperparametroPreditivo	0	20	0%
l = maxDepth	1	5	20%
m = conceitoArvoreDecisao	0	5	0%
n = desvantagem	3	30	10%
o = overfitting	3	30	10%
p = randomState	0	5	0%
q = comparacao	21	45	47%
r = historia	49	75	65%
s = oobScore	0	5	0%
t = conceitoBoosting	0	5	0%
u = hiperparametro	8	25	32%
v = conceitoArvore	0	5	0%
w = usabilidade	0	5	0%
x = nEstimators	5	20	25%
y = tarefa	1	5	20%
z = conceitoBootstrap	0	4	0%
aa = vantagem	5	30	17%
ab = minSampleLeaf	0	10	0%
ac = bagging	5	31	16%
ad = melhoria	5	10	50%
ae = aplicacao	144	220	65%
af = nJobs	0	5	0%
ag = funcionamento	71	151	47%
ah = implementacao	0	5	0%
ai = desbalanceamento	3	5	60%
aj = hiperparametros	0	10	0%
ak = hiperparametroVelocidade	3	20	15%
al = conceitoHiperparametro	0	5	0%
am = classificacao	0	5	0%

Observamos que não há uma relação direta entre a frequência de perguntas da classe no corpus e o índice de *recall*. Basta ver, por exemplo, que a classe aplicacao, a mais frequente no corpus (total de 220 ocorrências), teve desempenho inferior à classe posprocessamento (apenas 5 ocorrências e 100% de *recall*).