Search or jump to…

Pulls

Issues

Codespaces

Marketplace

Explore

@sriram3107

IBM-EPBL

/

IBM-Project-25245-1659955738

Public

Code

Issues

2

Pull requests

1

Actions

Projects

Security

Insights

IBM-Project-25245-1659955738/Project Development Phase/Sprint 2/functionalForJobRecommender.py.py /

@Prasan2002

Prasan2002 Create functionalForJobRecommender.py.py

Latest commit 5ee5bbb 2 days ago

History

1 contributor

316 lines (280 sloc) 16.2 KB

From functools import reduce

Import re

From nltk.corpus import stopwords

From sklearn.feature\_extraction.text import CountVectorizer

From sklearn.metrics.pairwise import cosine\_similarity

Import PyPDF2

Import pandas as pd

From sklearn.preprocessing import MinMaxScaler

Import matplotlib.pyplot as plt

From collections import Counter

Import numpy as np

Pd.options.mode.chained\_assignment = None

# Skill dictionary used for the project

SKillDictionary = [‘bash’, ‘r’, ‘python’, ‘java’, ‘c++’, ‘ruby’, ‘perl’, ‘matlab’, ‘javascript’, ‘scala’, ‘php’,

‘jquery’, ‘angularjs’, ‘excel’, ‘tableau’, ‘sas’, ‘spss’, ‘d3’, ‘saas’, ‘pandas’, ‘numpy’, ‘scipy’,

‘sps’, ‘spotfire’, ‘scikit’, ‘splunk’, ‘power’, ‘h2o’, ‘pytorch’, ‘tensorflow’, ‘caffe’, ‘caffe2’,

‘cntk’, ‘mxnet’, ‘paddle’, ‘keras’, ‘bigdl’, ‘hadoop’, ‘mapreduce’, ‘spark’, ‘pig’, ‘hive’, ‘shark’,

‘oozie’, ‘zookeeper’, ‘flume’, ‘mahout’, ‘etl’, ‘aws’, ‘azure’, ‘google’, ‘ibm’, ‘agile’, ‘devops’,

‘scrum’, ‘agile’, ‘devops’, ‘scrum’, ‘sql’, ‘nosql’, ‘hbase’, ‘cassandra’, ‘mongodb’, ‘mysql’,

‘mssql’, ‘postgresql’, ‘oracle’, ‘rdbms’, ‘bigquery’]

# creating a dataframe to add job description list

JobDescriptionDataframe = pd.DataFrame()

# class for job recommendation using dynamic weightage on Implicit and Explicit skills of Job description.

Class FunctionsForJobRecommendation:

# Init to convert job description list to a dataframe

Def \_\_init\_\_(self, jobs\_list):

Pd.set\_option(‘display.max\_columns’, None)

Pd.set\_option(‘display.max\_rows’, None)

Self.JobDescriptionDataframe = pd.DataFrame(jobs\_list)

# Function to extract keywords extracted and filtered by using Skill dictionary

Def ExtractKeywords(self, text):

Text = text.lower()

Text = re.sub(r”[()<>/]”, ‘, ‘, text) # substitute ()<>&/ to comma and space

Text = re.sub(r”&”, ‘and’, text) # substitute ()<>&/ to comma and space

Text = re.sub(r”[?!]”, ‘. ‘, text) # substitute ?! to dot and space

Text = re.sub(“ [a-z0-9]+[.’\-a-z0-9\_]\*[a-z0-9][+@\w+\.com](mailto:+@\w+\.com)”, “”, text) # substitute email address to dot

Text = re.sub(‘ +’, ‘ ‘, text) # replace multiple whitespace by one whitespace

Text = text.lower().split()

Stops = set(stopwords.words(“english”)) # Filter out stop words in english language

Text = [w for w in text if not w in stops]

Text = list(set(text))

# Skills are extracted from the preprocessed text

# keywords extracted and filtered by using Skill dictionary

Keywords = [str(word) for word in text if word in SKillDictionary]

Return Keywords

# Function to use counter to count the frequency of the keywords

Def CountKeywords(self, keywords, counter):

KeywordCount = pd.DataFrame(columns=[‘Freq’])

For EachWord in keywords:

KeywordCount.loc[EachWord] = {‘Freq’: counter[EachWord]}

Return KeywordCount

# Function to extract skill keywords from job description

Def ExtractJobDescKeywords(self):

# removing duplicate Jobs

Self.JobDescriptionDataframe.drop\_duplicates(subset=[‘desc’], inplace=True, keep=’last’, ignore\_index=False)

# Extract skill keywords from job descriptions and store them in a new column ‘keywords’

Self.JobDescriptionDataframe[‘keywords’] = [self.ExtractKeywords(job\_desc) for job\_desc in

Self.JobDescriptionDataframe[‘desc’]]

# Function to extract resume keywords from resume

Def ExtractResumeKeywords(self, resume\_pdf):

# Open resume PDF

Resume = open(resume\_pdf, ‘rb’)

# creating a pdf reader object

ReadResume = PyPDF2.PdfFileReader(Resume)

# Read in each page in PDF

ResumeContext = [ReadResume.getPage(x).extractText() for x in range(ReadResume.numPages)]

# Extract key skills from each page

ResumeKeywords = [self.ExtractKeywords(page) for page in ResumeContext]

# Count keywords

ResumeFrequency = Counter()

For item in ResumeKeywords:

ResumeFrequency.update(item)

# Get resume skill keywords counts

ResumeSkilllist = self.CountKeywords(SKillDictionary, ResumeFrequency)

Return ResumeSkilllist[ResumeSkilllist[‘Freq’] > 0]

# Cosine similarity function to calculate cosine score between two documents

Def CalculateCosineSimilarity(self, documents):

Countvectorizer = CountVectorizer()

Matrix = Countvectorizer.fit\_transform(documents)

DocumentMatrix = Matrix.todense()

Df = pd.DataFrame(DocumentMatrix,

Columns=Countvectorizer.get\_feature\_names(),

Index=[‘ind1’, ‘ind2’])

Return cosine\_similarity(df)[0][1]

# Function to calculate similarity and pick top10 jobs that match the resume

Def CalculateSimilarity(self, ResumeSkillList):

# copy of job description dataframe as JobDescriptionSet

JobDescriptionSet = self.JobDescriptionDataframe.copy()

# To calculate similarity between resume skills and skills extracted from job description

For ind, x in JobDescriptionSet.iterrows():

JobDescriptionString = ‘ ‘.join(map(str, x.keywords))

ResumeKeywordString = ‘ ‘.join(map(str, ResumeSkillList))

Documents = [JobDescriptionString, ResumeKeywordString]

# Created a column ‘cosinescore’ to store cosine score for top10 jobs

JobDescriptionSet.loc[ind, ‘cosinescore’] = self.CalculateCosineSimilarity(documents)

# to sort the top10 description based on cosine score

MainTop10JDs = JobDescriptionSet.sort\_values(by=’cosinescore’, ascending=False).head(10)

Return MainTop10JDs

# Function to extract top20 Job description for each of the top10 jobs to get implicit skills

Def Extract20SimilarJDs(self, dynStat, MainTop10JDs, ResumeSkillList):

JobDescriptionSet = self.JobDescriptionDataframe.copy()

SimilarJobIdsDataframe = pd.DataFrame()

SimilarJobIdsDataframe.loc[0, ‘similarJDs’] = ‘NaN’

Count2 = 0

finalSkillWeightList = []

# Iterate through each of the top 10 Jobs to extract similar 20 JDs

For ind, x in MainTop10JDs.iterrows():

# variables for GraphPlot function ##

impSkillCountResumeMatch = 0

ImpSkillWeightCount = 0

implicitSkillList = []

implicitSkillWeightList = []

# To extract each JD keyword set

PickedJobDescriptionString = ‘ ‘.join(map(str, x.keywords))

JDKeywordsSet = set(x.keywords)

# To pick the common skills between resume and TopJD and added them to exSkillCountResumeMatch list##

Intersection = JDKeywordsSet.intersection(ResumeSkillList)

exSkillCountResumeMatch = len(intersection)

# Variable declared to calculate 20 similar Job description for each of Top10 Jobs

Rows = []

Count2 = count2 + 1

# Iterate through the whole job description dataset to pick 20 similar Job description for each Top10 Jobs

For ind2, x2 in JobDescriptionSet.iterrows():

# To skip the topJD within the job description

If ind == ind2:

Continue

JobDescriptionString = ‘ ‘.join(map(str, x2.keywords))

# to calculate cosine score between topJD skills and pickedJD

Documents = [JobDescriptionString, PickedJobDescriptionString]

Rows.append([ind2, self.CalculateCosineSimilarity(documents)])

# create a dataframe column for each of 20 similar Jds to store their cosine score

SimilarJobIdsDataframe[‘JD’] = ind2

SimilarJobIdsDataframe[‘cosScore’] = self.CalculateCosineSimilarity(documents)

Rows.sort(key=lambda i: i[1], reverse=True)

Count = 0

JobDescriptionString = ‘ ‘

For row in rows:

Indexval = ‘JDind’ + str(count)

Count = count + 1

MainTop10JDs.loc[ind, indexval] = row[0]

JobDescriptionString = JobDescriptionString + ‘ ‘ + ‘ ‘.join(

Map(str, JobDescriptionSet.keywords[MainTop10JDs.at[ind, indexval]]))

# set a threshold to collect top20 JobIds for each of Top10Jobs

If count > 20:

Break

# Create a dataframe ‘skill\_list’ to store the implicit skills of top20 JDs for each top Job

MainTop10JDs.loc[ind, ‘skill\_list’] = JobDescriptionString

# Assign skill\_list to WordList to assign static and dynamic weightage.

WordList = MainTop10JDs.loc[ind, ‘skill\_list’]

WordList = WordList.split()

ImplicitWeight = 10

# For Graph plot function ####

skillList = []

for implicitSkill in np.unique(np.array(WordList)):

if implicitSkill in ResumeSkillList:

if implicitSkill not in x.keywords:

impSkillCountResumeMatch = impSkillCountResumeMatch + 1

# implicitSkillList is the list of implicit skills which are also present in resume

implicitSkillList.append(implicitSkill)

MainTop10JDs.loc[ind, ‘exSkillCountResumeMatch’] = exSkillCountResumeMatch

MainTop10JDs.loc[ind, ‘impSkillCountResumeMatch’] = impSkillCountResumeMatch

# for each implicit skill and its term frequency in the implicit skill list

For word, freq in Counter(WordList).items():

If word in MainTop10JDs.keywords[ind]:

Continue

# For dynamic approach, assign weightage based on term frequency. Higher the count of the term present in the skilllist, higher the weightage.

If (dynStat == 1):

tmpList = (word, freq / sum(Counter(WordList).values()) \* ImplicitWeight)

if word in implicitSkillList:

ImpSkillWeightCount = ImpSkillWeightCount + tmpList[1]

# For static appraoch, setting weight to 1 and disabling dynamic weight

Else:

tmpList = (word, 1)

if word in implicitSkillList:

ImpSkillWeightCount = ImpSkillWeightCount + tmpList[1]

skillList.append(tmpList)

# For Graph plot function

If dynStat == 1:

For skill, weight in skillList:

If skill in implicitSkillList:

implicitSkillWeightList.append((skill, weight))

finalSkillWeightList.append((ind, implicitSkillWeightList))

# Assign weightage of 1 to explicit skills for both static and dynamic approach

Top10keywords = MainTop10JDs.keywords[ind]

exSkillList = []

for skill in top10keywords:

tmpList = (skill, 1)

exSkillList.append(tmpList)

MainTop10JDs.keywords[ind] = exSkillList

MainTop10JDs.keywords[ind] = MainTop10JDs.keywords[ind] + skillList

Sorted(MainTop10JDs.keywords[ind], key=lambda x: x[1], reverse=True)

# top\_10\_jd\_matches – to return top10 Jobs with 20 similar JD for each top Job and their skill weightage.

# finalSkillWeightList – for Graph plot function, pick the implicit skills which match the resume along with its dynamic weightage.

Return MainTop10JDs, finalSkillWeightList

# Function to calculate final cosine score for each top Job using weighted cosine similarity and rank them according to the cosine score.

Def WeightedCosineSimilarity(self, ResumeSkillList, Implicit):

rsmSkillList = []

# adding wightage of 1 to resume skill list as they should be given high priority

For skill in ResumeSkillList:

rsmSkillList.append((skill, 1))

# For each of the Top 10 Jobs

For ind, x in Implicit.iterrows():

# Create one dictionary for resume skill list and another for job description skills(Implicit +explicit)

D1 = dict(rsmSkillList)

D2 = dict(Implicit.keywords[ind])

# Using weightage cosine similarity because the weightage differ based on term frequency for implicit skills in dynamic approach

Allkey = reduce(set.union, map(set, map(dict.keys, [d1, d2])))

V1 = np.zeros((len(allkey),))

K = 0

For I in allkey:

If I in d1.keys():

V1[k] = d1[i]

K = k + 1

V2 = np.zeros((len(allkey),))

K = 0

For I in allkey:

If I in d2.keys():

V2[k] = d2[i]

K = k + 1

# v1 and v2 are 1-d np arrays representing resume skill list and job description skills

V1 = (v1 / np.sqrt(np.dot(v1, v1))) ## normalized

V2 = (v2 / np.sqrt(np.dot(v2, v2))) ## normalized

Implicit.loc[ind, ‘final\_cosine’] = np.dot(v1, v2)

# sort values based on cosine score

Implicit = Implicit.sort\_values(by=’final\_cosine’, ascending=False)

Implicit.reset\_index(inplace=True)

Implicit = Implicit.rename(columns={‘index’: ‘Jobid’})

# return dataframe which consists of final cosine score calculated using dynamic weightage and ranked top10 JDs that best match the resume.

Return Implicit

# Function to plot graphs for evaluation of the proposed approach

Def AllGraphPlotsForEvaluation(self, StaticGraph, DynamicGraph, finalSkillWeightList, dynStat):

For dynStat in range(0, 2):

If (dynStat == 0):

ImplicitGraph = StaticGraph

Else:

ImplicitGraph = DynamicGraph

# create a scaler object for normalizing data points

Scaler = MinMaxScaler()

Df\_norm = pd.DataFrame(scaler.fit\_transform(ImplicitGraph), columns=ImplicitGraph.columns)

ImplicitGraph[‘final\_cosine’] = df\_norm[‘final\_cosine’]

# Scatter plot for graph showing difference in cosine score

Size = np.array([])

For x in ImplicitGraph[‘final\_cosine’]:

Size = np.append(size, x \* 1000)

Plt.scatter(x=ImplicitGraph[‘final\_cosine’], y=ImplicitGraph[‘Jobid’], s=size,

C=ImplicitGraph[‘final\_cosine’], cmap=’viridis’, alpha=0.5)

Plt.colorbar(label=’Normalized cosine score’)

# Creating comparitive bar plot for implicit and explicit skill count for referenced and proposed solution

# creating a list of all inputs:

# Jobid

# expcount- count of the explicit skills of the job description which match the resume

# impcount – count of implicit skills of the job description which match the resume

Index = ImplicitGraph[‘Jobid’].tolist()

expCount = ImplicitGraph[‘exSkillCountResumeMatch’].tolist()

impCount = ImplicitGraph[‘impSkillCountResumeMatch’].tolist()

df = pd.DataFrame({‘exSkillCountResumeMatch’: expCount, ‘impSkillCountResumeMatch’: impCount}, index=index)

ax = df.plot.bar(rot=0)

ax.set\_xlabel(‘Job ID’)

ax.set\_ylabel(‘Implicit\_and\_Explicit\_Resume\_match\_with\_Implicit’)

# Barplot for dynamic approach to show how the implicit skills weightage influence ranking of the job list.

Df2 = df

If (dynStat == 1):

Index = []

Df = pd.DataFrame()

indexNo = 0

for ind, skillList in finalSkillWeightList:

if not skillList:

continue

index.append(ind)

for skill, weight in skillList:

df.loc[indexNo, [skill]] = weight

indexNo = indexNo + 1

# print

Df.index = index

Df = df.reindex(index=df2.index)

Ax = df.plot.bar(rot=0)

Ax.set\_xlabel(‘Job ID’)

Ax.set\_ylabel(‘Implicit\_and\_Explicit\_Resume\_match\_with\_Implicit’)

Plt.show()

Plt.clf()