```
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", "", 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()
```