Resume Parsing and Ranking System

Problem Statements:

Many organizations and companies use resumes as a way to pre-screen candidates.
Reviewing and shortlisting all the requirements is a complex process that often leads people to make mistakes.
This study aims to facilitate the review of the process by collecting retrospective information about the job description (JD).

Dataset:

Resumes and job descriptions collected from Indeed.com and LinkedIn.
Also, job postings on assertivebox.com have been removed. It also provides information on candidate selection for further analysis.

Tools:

Django framework used to create the user interface.
BeautifulSoup in Python is used to extract data from websites.
NLTk and Spacy libraries are used to process files.
PDFMiner.six is used to convert pdf to text.
Terms of use:
The project was developed in 3 stages.

Stage 1

- Collect information from job postings and job postings on Naukri.com and LinkedIn.
- Data Maintenance
- Create dictionaries and publish lists. All resumes are vectorized.

- Create a search engine.
- The question is a job description from the company.
- Check the similarities between the job description and the resume.
- Finally, the search engine will list and display suitable candidates.
- Scope of development: Integrating NLP technology such as NER into the above content model and improving the ranking method.

Stage 2

Vectorization:

- With the help of indexes, we use the tf-idf method to create data vectors (tf and idf normalization).
- We obtain a 570 * 3500 dimensional matrix; where 570 is the number of data and 3500 is the length of each vector.

Querying:

- There are two types of questions here. We may request information based on key terms (e.g. skills, university, etc.) or job description information.
- For the last approach, we should load JD for all the pre-processing (like the first stage) and then vectorize the query.
- We built this into the Django framework.

Sorting:

- For sorting, cosine similarity is used to initially keep the top N records.

- The list of selected candidates is available for the competition. The data was also sorted using the binomial model.
- Both ranking methods are evaluated using the Precision@R metric.

User Interface:

- The user interface of the query is created with the help of Django framework. A place has also been created to continue uploading.
- Stage 3
- In Stages 1 and 2, we created a simple search engine to recover and reset JD. In Phase 3, we improved the model in three main aspects: vectorization, data extraction, and the ranking process.

Vectorization:

- Use GloVe vectorization to vectorize a word.
- The GloVe model is based on international statistics and common words.
- Therefore, it is more suitable than the tf-idf method.

Information retrieval:

- For the information retrieval part, we use the method called authentication to extract the information group from the target.
- We created a custom corpus and trained our model to fit the content of our dataset.
- This will help us capture better features.

Ranking:

- Finally, we use the combined strategy of classification and ranking for ranking.
- Using Naive Bayes, SVM and RNN models for classification and using BM25 and cosine similarity metrics for ranking.
- Data for ranking is selected taking into account the highest vote of the classification, and then the selected data is selected based on the average of 2 ranking scores.

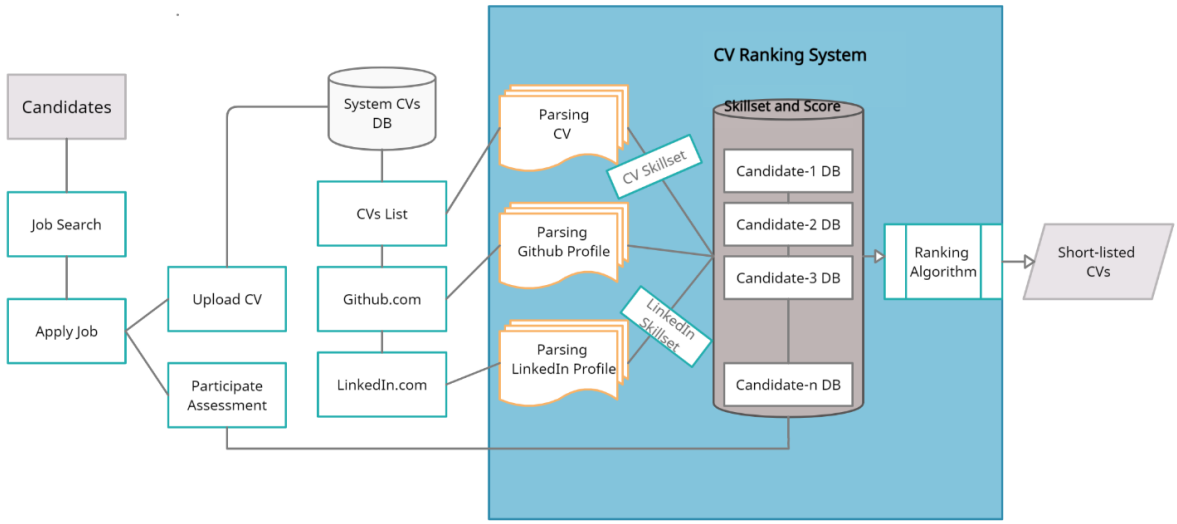
Evaluation:

- Evaluation of the model using Precision #R
- We achieved a Precision#R value of 52% for the model
- After correcting and using the research we increased the precision @ R to 81% .

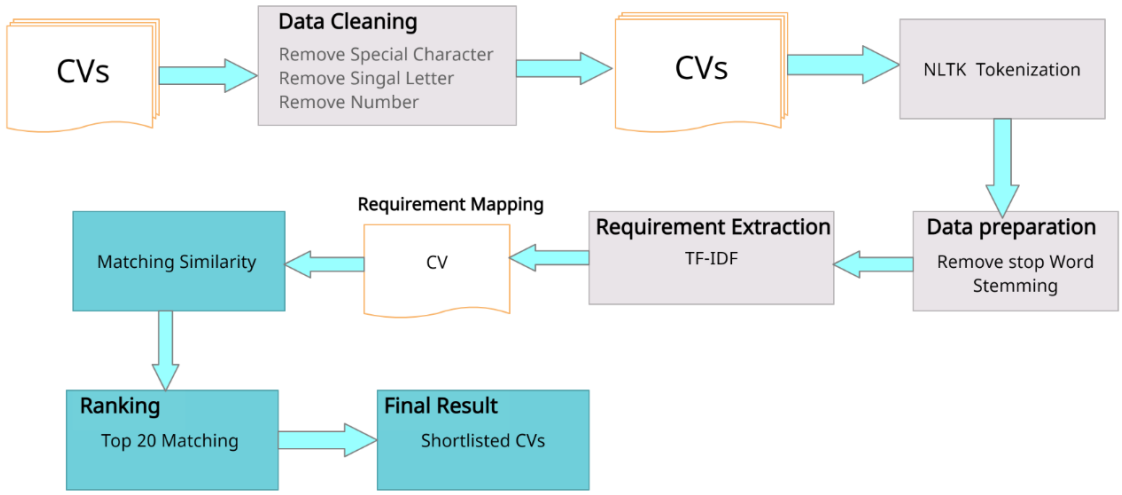
Create a template:

- Create a virtual environment for development dependencies. Creating venv.
- Install dependencies from Requirements.txt
- Create the Djando framework. Run Django.

System Architecture



System Model



Source code:

patterns and constants are designed to aid in the extraction and processing of information from resumes, such as identifying names, education details, dates, and different sections of the resume.

```
from nltk.corpus import stopwords

NAME_PATTERN = [{'POS': 'PROPN'}, {'POS': 'PROPN'}]

# Education (Upper Case Mandatory)
EDUCATION = [
        'BE', 'B.E.', 'B.E', 'BS', 'B.S', 'ME', 'M.E',
        'M.E.', 'MS', 'M.S', 'BTECH', 'MTECH',
        'SSC', 'HSC', 'CBSE', 'ICSE', 'X', 'XII'
    ]
```

```
NOT_ALPHA_NUMERIC = r'[^a-zA-Z\d]'

NUMBER = r'\d+'

# For finding date ranges
MONTHS_SHORT = r'''(jan)|(feb)|(mar)|(apr)|(may)|(jun)|(jul)
            |(aug)|(sep)|(oct)|(nov)|(dec)'''
MONTHS_LONG =
r'''(january)|(february)|(march)|(april)|(may)|(june)|(july)|

(august)|(september)|(october)|(november)|(december)'''
MONTH = r'(' + MONTHS_SHORT + r'|' + MONTHS_LONG + r')'
YEAR = r'(((20|19)(\d{2})))'

STOPWORDS = set(stopwords.words('english'))

RESUME_SECTIONS_PROFESSIONAL = [
            'experience',
            'education',
            'interests',
            'professional experience',
            'publications',
            'skills',
            'certifications',
            'objective',
            'career objective',
            'summary',
            'leadership'
          ]

RESUME_SECTIONS_GRAD = [
            'accomplishments',
            'experience',
            'education',
            'interests',
```

```
        'projects',
        'professional experience',
        'publications',
        'skills',
        'certifications',
        'objective',
        'career objective',
        'summary',
        'leadership'
```

for converting pdf to txt:

```
import sys
import logging
import six
import pdfminer.settings
pdfminer.settings.STRICT = False
import pdfminer.high_level
import pdfminer.layout
from pdfminer.image import ImageWriter
import argparse


def extract_text(files=[], outfile='-',
        _py2_no_more_posargs=None,
        no_laparams=False, all_texts=None, detect_vertical=None,
        word_margin=None, char_margin=None, line_margin=None,
boxes_flow=None,
        output_type='text', codec='utf-8', strip_control=False,
        maxpages=0, page_numbers=None, password="", scale=1.0,
rotation=0,
        layoutmode='normal', output_dir=None, debug=False,
        disable_caching=False, **other):
```

```
    if _py2_no_more_posargs is not None:
        raise ValueError("Many args")
    if not files:
        raise ValueError("Enter Filename")


    if not no_laparams:
        laparams = pdfminer.layout.LAParams()
        for param in ("all_texts", "detect_vertical", "word_margin",
"char_margin", "line_margin", "boxes_flow"):
            paramv = locals().get(param, None)
            if paramv is not None:
                setattr(laparams, param, paramv)
    else:
        laparams = None

    imagewriter = None
    if output_dir:
        imagewriter = ImageWriter(output_dir)

    if output_type == "text" and outfile != "-":
        for override, alttype in ((".htm", "html"), (".html", "html"),
(".xml", "xml"),(".tag", "tag") ):
            if outfile.endswith(override):
                output_type = alttype

    if outfile == "-":
        outfp = sys.stdout
        if outfp.encoding is not None:
            codec = 'utf-8'
    else:
        outfp = open(outfile, "wb")


    for fname in files:
```

```
        with open(fname, "rb") as fp:
            pdfminer.high_level.extract_text_to_fp(fp, **locals())
            fp.close()
    return outfp


def main(args=None):
    P = argparse.ArgumentParser()
    A = P.parse_args(args=args)
    if A.page_numbers:
        A.page_numbers = set([x-1 for x in A.page_numbers])
    if A.pagenos:
        A.page_numbers = set([int(x)-1 for x in A.pagenos.split(",")])

    imagewriter = None
    if A.output_dir:
        imagewriter = ImageWriter(A.output_dir)

    if six.PY2 and sys.stdin.encoding:
        A.password = A.password.decode(sys.stdin.encoding)

    if A.output_type == "text" and A.outfile != "-":
        for override, alttype in ((".htm",  "html"), (".html",
"html"),(".xml",  "xml" ),(".tag",  "tag" )):
            if A.outfile.endswith(override):
                A.output_type = alttype

    if A.outfile == "-":
        outfp = sys.stdout
        if outfp.encoding is not None:
            A.codec = 'utf-8'
    else:
        outfp = open(A.outfile, "wb")

    outfp = extract_text(**vars(A))
```

```
    outfp.close()
    return 0


if __name__ == '__main__': sys.exit(main())
```

compare them against job descriptions, and rank the candidates based on their suitability for the job.

```
import warnings
import textract
import re
from sklearn.feature_extraction.text import CountVectorizer,
TfidfTransformer
from sklearn.neighbors import NearestNeighbors
import PyPDF2
from json import load, dumps
from operator import getitem
from collections import OrderedDict
from .text_process import normalize
from nltk.tokenize import word_tokenize
import mysite.configurations as regex
from datetime import date

from collections import defaultdict
from datetime import datetime
from dateutil import relativedelta
from typing import *


warnings.filterwarnings(action='ignore', category=UserWarning,
module='gensim')
```

```
def getFilePath(loc):
    temp = str(loc)
    temp = temp.replace('\\', '/')
    return temp


def getFileName(filename):
    return filename.rsplit('\\')[1]


def readResultInJson(jobfile='job1'):
    filepath = 'result/'
    with open(filepath + jobfile + '.json', 'r') as openfile:
        # Reading from json file
        result = load(openfile)
    return result


def writeResultInJson(data, jobfile='job1'):
    filepath = 'result/'
    json_str = dumps(data, indent=4)
    with open(filepath + jobfile + '.json', 'w+', encoding='utf-8') as f:
        f.write(json_str)
        f.close()


def getNumberOfMonths(datepair) -> int:
    """
    Helper function to extract total months of experience from a
resume
    :param date1: Starting date
    :param date2: Ending date
    :return: months of experience from date1 to date2
```

```
    """
# if years
# if years
date2_parsed = False
if datepair.get("fh", None) is not None:
    gap = datepair["fh"]
else:
    gap = ""
try:
    present_vocab = ("present", "date", "now")
    if "syear" in datepair:
        date1 = datepair["fyear"]
        date2 = datepair["syear"]

        if date2.lower() in present_vocab:
            date2 = datetime.now()
            date2_parsed = True

        try:
            if not date2_parsed:
                date2 = datetime.strptime(str(date2), "%Y")
            date1 = datetime.strptime(str(date1), "%Y")
        except:
            pass
    elif "smonth_num" in datepair:
        date1 = datepair["fmonth_num"]
        date2 = datepair["smonth_num"]

        if date2.lower() in present_vocab:
            date2 = datetime.now()
            date2_parsed = True

        for stype in ("%m" + gap + "%Y", "%m" + gap + "%y"):
            try:
                if not date2_parsed:
```

```
                date2 = datetime.strptime(str(date2), stype)
            date1 = datetime.strptime(str(date1), stype)
            break
         except:
            pass
      else:
         date1 = datepair["fmonth"]
         date2 = datepair["smonth"]

         if date2.lower() in present_vocab:
            date2 = datetime.now()
            date2_parsed = True

         for stype in (
            "%b" + gap + "%Y",
            "%b" + gap + "%y",
            "%B" + gap + "%Y",
            "%B" + gap + "%y",
         ):
            try:
               if not date2_parsed:
                  date2 = datetime.strptime(str(date2), stype)
               date1 = datetime.strptime(str(date1), stype)
               break
            except:
               pass

      months_of_experience = relativedelta.relativedelta(date2,
date1)
      months_of_experience = (
         months_of_experience.years * 12 +
months_of_experience.months
      )
      return months_of_experience
   except Exception as e:
```

```
        return 0


def getTotalExperience(experience_list) -> int:
    """
    Wrapper function to extract total months of experience from a
resume
    :param experience_list: list of experience text extracted
    :return: total months of experience
    """
    exp_ = []
    for line in experience_list:
        line = line.lower().strip()
        # have to split search since regex OR does not capture on a first-
come-first-serve basis
        experience = re.search(
            r"(?P<fyear>\d{4})\s*(\s|-
|to)\s*(?P<syear>\d{4}|present|date|now)",
            line,
            re.I,
        )
        if experience:
            d = experience.groupdict()
            exp_.append(d)
            continue

        experience = re.search(
            r"(?P<fmonth>\w+(?P<fh>.)\d+)\s*(\s|-
|to)\s*(?P<smonth>\w+(?P<sh>.)\d+|present|date|now)",
            line,
            re.I,
        )
        if experience:
            d = experience.groupdict()
            exp_.append(d)
```

```
            continue

        experience = re.search(
            r"(?P<fmonth_num>\d+(?P<fh>.)\d+)\s*(\s|-
|to)\s*(?P<smonth_num>\d+(?P<sh>.)\d+|present|date|now)",
            line,
            re.I,
        )
        if experience:
            d = experience.groupdict()
            exp_.append(d)
            continue
    experience_num_list = [getNumberOfMonths(i) for i in exp_]
    total_experience_in_months = sum(experience_num_list)
    return total_experience_in_months


"""
Utility Function that calculates experience in the resume text
params: resume_text type:string
returns: experience type:int
"""
def calculate_experience(resume_text):
  #
  def get_month_index(month):
    month_dict = {'jan':1, 'feb':2, 'mar':3, 'apr':4, 'may':5, 'jun':6, 'jul':7,
'aug':8, 'sep':9, 'oct':10, 'nov':11, 'dec':12}
    return month_dict[month.lower()]

  try:
    experience = 0
    start_month = -1
    start_year = -1
    end_month = -1
    end_year = -1
```

```
    regular_expression = re.compile(regex.date_range,
re.IGNORECASE)
    regex_result = re.search(regular_expression, resume_text)
    while regex_result:
      date_range = regex_result.group()
      year_regex = re.compile(regex.year)
      year_result = re.search(year_regex, date_range)
      if (start_year == -1) or (int(year_result.group()) <= start_year):
        start_year = int(year_result.group())
        month_regex = re.compile(regex.months_short,
re.IGNORECASE)
        month_result = re.search(month_regex, date_range)
        if month_result:
          current_month = get_month_index(month_result.group())
          if (start_month == -1) or (current_month < start_month):
            start_month = current_month
      if date_range.lower().find('present') != -1:
        end_month = date.today().month # current month
        end_year = date.today().year # current year
      else:
        year_result = re.search(year_regex,
date_range[year_result.end():])
        if (end_year == -1) or (int(year_result.group()) >= end_year):
          end_year = int(year_result.group())
          month_regex = re.compile(regex.months_short,
re.IGNORECASE)
          month_result = re.search(month_regex, date_range)
          if month_result:
            current_month = get_month_index(month_result.group())
            if (end_month == -1) or (current_month > end_month):
              end_month = current_month
      resume_text = resume_text[regex_result.end():]
      regex_result = re.search(regular_expression, resume_text)

    return end_year - start_year  # Use the obtained month attribute
```

```
  except Exception as exception_instance:
    # logging.error('Issue calculating experience:
'+str(exception_instance))
    print('Issue calculating experience: '+str(exception_instance))
    return None


def get_experience_year(job_expr):
    job_expr = str.split(job_expr, ' ')[0]
    if '-' in job_expr:
        expr = job_expr.split('-')
        return int(expr[0])*12, int(expr[1])*12
    return int(job_expr)*12, -1


# for 2nd method
def getTotalExperienceFormatted(exp_list, job_expr) -> bool:

# for 1st method
# def getTotalExperienceFormatted(text, job_expr) -> bool:

    # for 2nd method
    min_yr_in_month, max_yr_in_month =
get_experience_year(job_expr)
    print(min_yr_in_month, max_yr_in_month)
    print(exp_list)
    months = getTotalExperience(exp_list)

    # for 1st mehtod
    # months = 0
    # for line in text.split("\n"):
    #     line = re.sub(r"\s+", " ", line).strip()
    #     match = re.search(r"^.*:", line)
    #     if match:
    #         months += calculate_experience(line)
```

```
    #
    # months = calculate_experience(text)
    #
    #
    # entities = utils.extract_entity_sections_grad(text)
    # months =
round(utils.get_total_experience(entities['experience']) / 12, 2)
    # print(months)

    # for 2nd method
    if max_yr_in_month != -1:
        if (months >= min_yr_in_month) and (months <=
max_yr_in_month):
            return True
    else:
        if months >= min_yr_in_month:
            return True
    return False



    # if months < 12:
    #     return str(months) + " months"
    # years = months // 12
    # months = months % 12
    # return str(years) + " years " + str(months) + " months"


def findWorkAndEducation(text, name) -> Dict[str, List[str]]:
    categories = {"Work": ["(Work|WORK)",
"(Experience(s?)|EXPERIENCE(S?))", "(History|HISTORY)"]}
    inv_data = {v[0][1]: (v[0][0], k) for k, v in categories.items()}
    line_count = 0
    exp_list = defaultdict(list)
    name = name.lower()
```

```
    current_line = None
    is_dot = False
    is_space = True
    continuation_sent = []
    first_line = None
    unique_char_regex = "[^\sA-Za-z0-9\.\/\(\)\,\-\|]+"

    for line in text.split("\n"):
        line = re.sub(r"\s+", " ", line).strip()
        match = re.search(r"^.*:", line)
        if match:
            line = line[match.end():].strip()

        # get first non-space line for filtering since
        # sometimes it might be a page header
        if line and first_line is None:
            first_line = line

        # update line_countfirst since there are `continue`s below
        line_count += 1
        if (line_count - 1) in inv_data:
            current_line = inv_data[line_count - 1][1]
        # contains a full-blown state-machine for filtering stuff
        elif current_line == "Work":
            if line:
                # if name is inside, skip
                if name == line:
                    continue
                # if like first line of resume, skip
                if line == first_line:
                    continue
                # check if it's not a list with some unique character as list
bullet
                has_dot = re.findall(unique_char_regex, line[:5])
                # if last paragraph is a list item
```

```
            if is_dot:
                # if this paragraph is not a list item and the previous line is
a space
                if not has_dot and is_space:
                    if line[0].isupper() or re.findall(r"^\d+\.", line[:5]):
                        exp_list[current_line].append(line)
                        is_dot = False

            else:
                if not has_dot and (
                    line[0].isupper() or re.findall(r"^\d+\.", line[:5])
                ):
                    exp_list[current_line].append(line)
                    is_dot = False
            if has_dot:
                is_dot = True
            is_space = False
        else:
            is_space = True
    elif current_line == "Education":
        if line:
            # if not like first line
            if line == first_line:
                continue
            line = re.sub(unique_char_regex, '', line[:5]) + line[5:]
            if len(line) < 12:
                continuation_sent.append(line)
            else:
                if continuation_sent:
                    continuation_sent.append(line)
                    line = " ".join(continuation_sent)
                    continuation_sent = []
                exp_list[current_line].append(line)

return exp_list
```

```
def check_basicRequirement(resumes_data, job_data):
    # print(job_experience)
    Ordered_list_Resume = []
    Resumes = []
    Temp_pdf = []

    # filter resumes based on the gender
    if job_data.gender == 'Male':
        resumes_data = resumes_data.filter(gender='Male')
    elif job_data.gender == 'Female':
        resumes_data = resumes_data.filter(gender='Female')



    # resumes file path
    filepath = 'media/'



    resumes = [str(item.cv) for item in resumes_data]
    resumes_new = [item.split(':')[0] for item in resumes]
    resumes_new = [item for item in resumes_new if item != '']

    LIST_OF_FILES = resumes_new

    print("Total Files to Parse\t", len(LIST_OF_FILES))
    print("######## PARSING ########")
    for indx, file in enumerate(LIST_OF_FILES):
        Ordered_list_Resume.append(file)
        Temp = file.split('.')

        if Temp[1] == "pdf" or Temp[1] == "Pdf" or Temp[1] == "PDF":
            try:
                # print("This is PDF", indx)
                with open(filepath + file, 'rb') as pdf_file:
```

```
                # read_pdf = PyPDF2.PdfFileReader(pdf_file)
                read_pdf = PyPDF2.PdfFileReader(pdf_file, strict=False)

                number_of_pages = read_pdf.getNumPages()
                for page_number in range(number_of_pages):
                    page = read_pdf.getPage(page_number)
                    page_content = page.extractText()
                    page_content = page_content.replace('\n', '
').replace('\f', '').replace('\\uf[0-9]+',
                                                                 '').replace(
                        '\\u[0-9]+', '').replace('\\ufb[0-9]+', '')
                    # page_content.replace("\r", "")

                    Temp_pdf = str(Temp_pdf) + str(page_content)
                    # print(Temp_pdf)

                # 1st method
                # if
getTotalExperienceFormatted(findWorkAndEducation(Temp_pdf,
'Work'), job_data.experience):

                # 2nd method
                if getTotalExperienceFormatted(Temp_pdf,
job_data.experience):
                    # print('True')
                    Resumes.extend([Temp_pdf])
                # Resumes.extend([Temp_pdf])
                Temp_pdf = ''


                # f = open(str(i)+str("+") , 'w')
                # f.write(page_content)
                # f.close()
            except Exception as e:
                print(e)
```

```
        if Temp[1] == "doc" or Temp[1] == "Doc" or Temp[1] == "DOC":
            # print("This is DOC", file)

            try:
                a = textract.process(filepath)
                a = a.replace(b'\n', b' ')
                a = a.replace(b'\r', b' ')
                b = str(a)
                c = [b]
                Resumes.extend(c)
            except Exception as e:
                print(e)

        if Temp[1] == "docx" or Temp[1] == "Docx" or Temp[1] ==
"DOCX":
            # print("This is DOCX", file)
            try:
                a = textract.process(filepath + file)
                a = a.replace(b'\n', b' ')
                a = a.replace(b'\r', b' ')
                b = str(a)
                c = [b]
                Resumes.extend(c)
            except Exception as e:
                print(e)

        if Temp[1] == "exe" or Temp[1] == "Exe" or Temp[1] == "EXE":
            # print("This is EXE", file)
            pass
    print("Done Parsing.")

    return Resumes, Ordered_list_Resume
```

```
def get_rank(result_dict=None):

    if result_dict == None:
        return {}

    # new_result_dict = sorted(result_dict.items(), key=lambda item:
float(item[1]["score"]), reverse=False)
    new_result_dict = OrderedDict(sorted(result_dict.items(),
key=lambda item: getitem(item[1], 'score'), reverse=False))
    new_updated_result_dict = {}
    indx = 0
    for _, item in new_result_dict.items():
        item['rank'] = indx + 1
        new_updated_result_dict[indx] = item
        indx += 1
    return new_updated_result_dict


def show_rank(result_dict=None, jobfileName='job1', top_k=20):
    if (result_dict == None):
        filepath = 'result/' + jobfileName + '.json'
        result_dict = readResultInJson(filepath)
    print("\nResult:")
    for _, result in result_dict.items():
        # print(result)
        print(f"Rank: {result['rank']}\t Total Score:{round(result['score'],
5)} (NN distance) \tName:{result['name']}")


# start parsing
# result
def res(resumes_data, job_data):

    # checking basic requirements
```

```
    Resumes, Ordered_list_Resume =
check_basicRequirement(resumes_data, job_data)


    # job-description

    Job_Desc = 0
    job_desc_filepath = 'jobDetails/'
    jobfilename = job_data.company_name + '_' + job_data.title + '.txt'
    job_desc = job_data.details + '\n' + job_data.responsibilities + '\n'
+ job_data.experience + '\n';
    job_desc = re.sub(r' +', ' ', job_desc.replace('\n', '').replace('\r', ''))

    try:
        text = re.sub(' +', ' ', job_desc)
        tttt = str(text)
        tttt = normalize(word_tokenize(tttt))
        text = [' '.join(tttt)]
    except:
        text = 'None'
    print("\nNormalized Job Description:\n", text)


    # get tf-idf of Job Description
    vectorizer = CountVectorizer(stop_words='english')
    transformar = TfidfTransformer()
    vectorizer.fit(text)
    vector =
transformar.fit_transform(vectorizer.transform(text).toarray())
    Job_Desc = vector.toarray()
    print("\nTF-IDF weight  (For Job Description):\n", Job_Desc, '\n')

    # get TF-IDF of Candidate Resumes
    Resume_Vector = []
    for file in Resumes:
```

```
        text = file
        tttt = str(text)
        try:

            tttt = normalize(word_tokenize(tttt))
            text = [' '.join(tttt)]

            vector =
transformar.fit_transform(vectorizer.transform(text).toarray())

            aaa = vector.toarray()
            print("TF-IDF weight(For Resumes): \n", aaa)
            Resume_Vector.append(aaa)
        except:
            pass

    # ranking process
    result_arr = dict()
    for indx, file in enumerate(Resume_Vector):
        samples = file
        name = Ordered_list_Resume[indx]
        neigh = NearestNeighbors(n_neighbors=1)
        neigh.fit(samples)
        NearestNeighbors(algorithm='auto', leaf_size=30)

        # score = round(neigh.kneighbors(Job_Desc)[0][0][0], 5)
        score = neigh.kneighbors(Job_Desc)[0][0][0]
        # print(score)
        result_arr[indx] = {'name': name, 'score': score}

    result_arr = get_rank(result_arr)
    # writeResultInJson(result_arr, jobfilename)
    show_rank(result_arr, jobfilename)

    # return resultant shortlist
```

```
    return result_arr
```

extracting information from resumes.

```
import io
import os
import re
import nltk
import pandas as pd
import docx2txt
from datetime import datetime
from dateutil import relativedelta
from extra import constants as cs
from pdfminer.converter import TextConverter
from pdfminer.pdfinterp import PDFPageInterpreter
from pdfminer.pdfinterp import PDFResourceManager
from pdfminer.layout import LAParams
from pdfminer.pdfpage import PDFPage
from pdfminer.pdfparser import PDFSyntaxError
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords


def extract_text_from_pdf(pdf_path):
    '''
    Helper function to extract the plain text from .pdf files
    :param pdf_path: path to PDF file to be extracted (remote or local)
    :return: iterator of string of extracted text
    '''
    # https://www.blog.pythonlibrary.org/2018/05/03/exporting-
data-from-pdfs-with-python/
    if not isinstance(pdf_path, io.BytesIO):
        # extract text from local pdf file
        with open(pdf_path, 'rb') as fh:
            try:
```

```
            for page in PDFPage.get_pages(
                    fh,
                    caching=True,
                    check_extractable=True
            ):
                resource_manager = PDFResourceManager()
                fake_file_handle = io.StringIO()
                converter = TextConverter(
                    resource_manager,
                    fake_file_handle,
                    codec='utf-8',
                    laparams=LAParams()
                )
                page_interpreter = PDFPageInterpreter(
                    resource_manager,
                    converter
                )
                page_interpreter.process_page(page)

                text = fake_file_handle.getvalue()
                yield text

                # close open handles
                converter.close()
                fake_file_handle.close()
        except PDFSyntaxError:
            return
    else:
        # extract text from remote pdf file
        try:
            for page in PDFPage.get_pages(
                    pdf_path,
                    caching=True,
                    check_extractable=True
            ):
```

```
            resource_manager = PDFResourceManager()
            fake_file_handle = io.StringIO()
            converter = TextConverter(
               resource_manager,
               fake_file_handle,
               codec='utf-8',
               laparams=LAParams()
            )
            page_interpreter = PDFPageInterpreter(
               resource_manager,
               converter
            )
            page_interpreter.process_page(page)

            text = fake_file_handle.getvalue()
            yield text

            # close open handles
            converter.close()
            fake_file_handle.close()
      except PDFSyntaxError:
         return


def get_number_of_pages(file_name):
   try:
      if isinstance(file_name, io.BytesIO):
         # for remote pdf file
         count = 0
         for page in PDFPage.get_pages(
                  file_name,
                  caching=True,
                  check_extractable=True
         ):
            count += 1
```

```
            return count
        else:
            # for local pdf file
            if file_name.endswith('.pdf'):
                count = 0
                with open(file_name, 'rb') as fh:
                    for page in PDFPage.get_pages(
                            fh,
                            caching=True,
                            check_extractable=True
                    ):
                        count += 1
                return count
            else:
                return None
    except PDFSyntaxError:
        return None


def extract_text_from_docx(doc_path):
    '''
    Helper function to extract plain text from .docx files
    :param doc_path: path to .docx file to be extracted
    :return: string of extracted text
    '''
    try:
        temp = docx2txt.process(doc_path)
        text = [line.replace('\t', ' ') for line in temp.split('\n') if line]
        return ' '.join(text)
    except KeyError:
        return ' '


def extract_text_from_doc(doc_path):
    '''
```

```
    Helper function to extract plain text from .doc files
    :param doc_path: path to .doc file to be extracted
    :return: string of extracted text
    '''
    try:
        try:
            import textract
        except ImportError:
            return ' '
        text = textract.process(doc_path).decode('utf-8')
        return text
    except KeyError:
        return ' '


def extract_text(file_path, extension):
    '''
    Wrapper function to detect the file extension and call text
    extraction function accordingly
    :param file_path: path of file of which text is to be extracted
    :param extension: extension of file file_name
    '''
    text = ''
    if extension == '.pdf':
        for page in extract_text_from_pdf(file_path):
            text += ' ' + page
    elif extension == '.docx':
        text = extract_text_from_docx(file_path)
    elif extension == '.doc':
        text = extract_text_from_doc(file_path)
    return text


def extract_entity_sections_grad(text):
    '''
```

```
    Helper function to extract all the raw text from sections of
    resume specifically for graduates and undergraduates
    :param text: Raw text of resume
    :return: dictionary of entities
    '''
    text_split = [i.strip() for i in text.split('\n')]
    # sections_in_resume = [i for i in text_split if i.lower() in sections]
    entities = {}
    key = False
    for phrase in text_split:
        if len(phrase) == 1:
            p_key = phrase
        else:
            p_key = set(phrase.lower().split()) &
set(cs.RESUME_SECTIONS_GRAD)
        try:
            p_key = list(p_key)[0]
        except IndexError:
            pass
        if p_key in cs.RESUME_SECTIONS_GRAD:
            entities[p_key] = []
            key = p_key
        elif key and phrase.strip():
            entities[key].append(phrase)

    # entity_key = False
    # for entity in entities.keys():
    #     sub_entities = {}
    #     for entry in entities[entity]:
    #         if u'\u2022' not in entry:
    #             sub_entities[entry] = []
    #             entity_key = entry
    #         elif entity_key:
    #             sub_entities[entity_key].append(entry)
    #     entities[entity] = sub_entities
```

```
    # pprint.pprint(entities)

    # make entities that are not found None
    # for entity in cs.RESUME_SECTIONS:
    #     if entity not in entities.keys():
    #         entities[entity] = None
    return entities


def extract_entities_wih_custom_model(custom_nlp_text):
    '''
    Helper function to extract different entities with custom
    trained model using SpaCy's NER
    :param custom_nlp_text: object of spacy.tokens.doc.Doc
    :return: dictionary of entities
    '''
    entities = {}
    for ent in custom_nlp_text.ents:
        if ent.label_ not in entities.keys():
            entities[ent.label_] = [ent.text]
        else:
            entities[ent.label_].append(ent.text)
    for key in entities.keys():
        entities[key] = list(set(entities[key]))
    return entities


def get_total_experience(experience_list):
    '''
    Wrapper function to extract total months of experience from a
resume
    :param experience_list: list of experience text extracted
    :return: total months of experience
    '''
```

```
    exp_ = []
    for line in experience_list:
        experience = re.search(

r'(?P<fmonth>\w+.\d+)\s*(\D|to)\s*(?P<smonth>\w+.\d+|present)',
            line,
            re.I
        )
        if experience:
            exp_.append(experience.groups())
    total_exp = sum(
        [get_number_of_months_from_dates(i[0], i[2]) for i in exp_]
    )
    total_experience_in_months = total_exp
    return total_experience_in_months


def get_number_of_months_from_dates(date1, date2):
    '''
    Helper function to extract total months of experience from a
resume
    :param date1: Starting date
    :param date2: Ending date
    :return: months of experience from date1 to date2
    '''
    if date2.lower() == 'present':
        date2 = datetime.now().strftime('%b %Y')
    try:
        if len(date1.split()[0]) > 3:
            date1 = date1.split()
            date1 = date1[0][:3] + ' ' + date1[1]
        if len(date2.split()[0]) > 3:
            date2 = date2.split()
            date2 = date2[0][:3] + ' ' + date2[1]
    except IndexError:
```

```
        return 0
    try:
        date1 = datetime.strptime(str(date1), '%b %Y')
        date2 = datetime.strptime(str(date2), '%b %Y')
        months_of_experience = relativedelta.relativedelta(date2,
date1)
        months_of_experience = (months_of_experience.years
                                * 12 + months_of_experience.months)
    except ValueError:
        return 0
    return months_of_experience


def extract_entity_sections_professional(text):
    '''
    Helper function to extract all the raw text from sections of
    resume specifically for professionals
    :param text: Raw text of resume
    :return: dictionary of entities
    '''
    text_split = [i.strip() for i in text.split('\n')]
    entities = {}
    key = False
    for phrase in text_split:
        if len(phrase) == 1:
            p_key = phrase
        else:
            p_key = set(phrase.lower().split()) \
                    & set(cs.RESUME_SECTIONS_PROFESSIONAL)
        try:
            p_key = list(p_key)[0]
        except IndexError:
            pass
        if p_key in cs.RESUME_SECTIONS_PROFESSIONAL:
            entities[p_key] = []
```

```
            key = p_key
        elif key and phrase.strip():
            entities[key].append(phrase)
    return entities


def extract_email(text):
    '''
    Helper function to extract email id from text
    :param text: plain text extracted from resume file
    '''
    email = re.findall(r"([^@|\s]+@[^@]+\.[^@|\s]+)", text)
    if email:
        try:
            return email[0].split()[0].strip(';')
        except IndexError:
            return None


def extract_name(nlp_text, matcher):
    '''
    Helper function to extract name from spacy nlp text
    :param nlp_text: object of spacy.tokens.doc.Doc
    :param matcher: object of spacy.matcher.Matcher
    :return: string of full name
    '''
    pattern = [cs.NAME_PATTERN]

    matcher.add('NAME', None, *pattern)

    matches = matcher(nlp_text)

    for _, start, end in matches:
        span = nlp_text[start:end]
        if 'name' not in span.text.lower():
```

```
        return span.text


def extract_mobile_number(text, custom_regex=None):
    '''
    Helper function to extract mobile number from text
    :param text: plain text extracted from resume file
    :return: string of extracted mobile numbers
    '''
    # Found this complicated regex on :
    # https://zapier.com/blog/extract-links-email-phone-regex/
    # mob_num_regex = r'''(?:(?:\+?([1-9]|[0-9][0-9]|
    #     [0-9][0-9][0-9])\s*(?:[.-]\s*)?)?(?:\(\s*([2-9]1[02-9]|
    #     [2-9][02-8]1|[2-9][02-8][02-9])\s*\)|([0-9][1-9]|
    #     [0-9]1[02-9]|[2-9][02-8]1|
    #     [2-9][02-8][02-9]))\s*(?:[.-]\s*)?)?([2-9]1[02-9]|
    #     [2-9][02-9]1|[2-9][02-9]{2})\s*(?:[.-]\s*)?([0-9]{7})
    #     (?:\s*(?:#|x\.?|ext\.?|
    #     extension)\s*(\d+))?'''
    if not custom_regex:
        mob_num_regex = r'''(\d{3}[-\.\s]??\d{3}[-\.\s]??\d{4}|\(\d{3}\)
                        [-\.\s]*\d{3}[-\.\s]??\d{4}|\d{3}[-\.\s]??\d{4})'''
        phone = re.findall(re.compile(mob_num_regex), text)
    else:
        phone = re.findall(re.compile(custom_regex), text)
    if phone:
        number = ''.join(phone[0])
        return number


def extract_skills(nlp_text, noun_chunks, skills_file=None):
    '''
    Helper function to extract skills from spacy nlp text
    :param nlp_text: object of spacy.tokens.doc.Doc
    :param noun_chunks: noun chunks extracted from nlp text
```

```
    :return: list of skills extracted
    '''
    tokens = [token.text for token in nlp_text if not token.is_stop]
    if not skills_file:
        data = pd.read_csv(
            os.path.join(os.path.dirname(_file_), 'skills.csv')
        )
    else:
        data = pd.read_csv(skills_file)
    skills = list(data.columns.values)
    skillset = []
    # check for one-grams
    for token in tokens:
        if token.lower() in skills:
            skillset.append(token)

    # check for bi-grams and tri-grams
    for token in noun_chunks:
        token = token.text.lower().strip()
        if token in skills:
            skillset.append(token)
    return [i.capitalize() for i in set([i.lower() for i in skillset])]


def cleanup(token, lower=True):
    if lower:
        token = token.lower()
    return token.strip()


def extract_education(nlp_text):
    '''
    Helper function to extract education from spacy nlp text
    :param nlp_text: object of spacy.tokens.doc.Doc
    :return: tuple of education degree and year if year if found
```

```
        else only returns education degree
    '''
    edu = {}
    # Extract education degree
    try:
        for index, text in enumerate(nlp_text):
            for tex in text.split():
                tex = re.sub(r'[?|$|.|!|,]', r'', tex)
                if tex.upper() in cs.EDUCATION and tex not in
cs.STOPWORDS:
                    edu[tex] = text + nlp_text[index + 1]
    except IndexError:
        pass

    # Extract year
    education = []
    for key in edu.keys():
        year = re.search(re.compile(cs.YEAR), edu[key])
        if year:
            education.append((key, ''.join(year.group(0))))
        else:
            education.append(key)
    return education


def extract_experience(resume_text):
    '''
    Helper function to extract experience from resume text
    :param resume_text: Plain resume text
    :return: list of experience
    '''
    wordnet_lemmatizer = WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))

    # word tokenization
```

```
word_tokens = nltk.word_tokenize(resume_text)

# remove stop words and lemmatize
filtered_sentence = [
       w for w in word_tokens if w not
       in stop_words and wordnet_lemmatizer.lemmatize(w)
       not in stop_words
   ]
sent = nltk.pos_tag(filtered_sentence)

# parse regex
cp = nltk.RegexpParser('P: {<NNP>+}')
cs = cp.parse(sent)

# for i in cs.subtrees(filter=lambda x: x.label() == 'P'):
#     print(i)

test = []

for vp in list(
   cs.subtrees(filter=lambda x: x.label() == 'P')
):
   test.append(" ".join([
      i[0] for i in vp.leaves()
      if len(vp.leaves()) >= 2])
   )

# Search the word 'experience' in the chunk and
# then print out the text after it
x = [
   x[x.lower().index('experience') + 10:]
   for i, x in enumerate(test)
   if x and 'experience' in x.lower()
]
```

```
return x
```