Vec2Rec – a Recommendation Engine for Job Seekers, Headhunters and Training Professionals Alike!

Author – Choi, Kwong Chak Michael

Table of Contents

[Goal 2](#_Toc39520657)

[Use cases 2](#_Toc39520658)

[Architecture 3](#_Toc39520659)

[Components 3](#_Toc39520660)

[Batch processing – to produce Gensim Doc2Vec models for similarity lookups 3](#_Toc39520661)

[Front end – for Similarity Queries 4](#_Toc39520662)

[Package Structure 5](#_Toc39520663)

[vec2rec.preprocess.tools 5](#_Toc39520664)

[vec2rec.frontend.vec2vec 5](#_Toc39520665)

[vec2rec.models.nlpmodels 6](#_Toc39520666)

[vec2rec.kfp.pipeline 7](#_Toc39520667)

[Technology Stack 7](#_Toc39520668)

[Design Choices & Implementation Considerations 7](#_Toc39520669)

[1) Preserve computation steps to allow easy reruns under different config / data updates 7](#_Toc39520670)

[2) Modularize components and enable future enhancement / replacement 7](#_Toc39520671)

[3) Enhance parallelization on computation and Memory Efficiency 7](#_Toc39520672)

[4) Allow incremental growth to lookup database 7](#_Toc39520673)

[5) Centralized Repository 8](#_Toc39520674)

[Future Enhancements 8](#_Toc39520675)

[References 8](#_Toc39520676)

# Goal

All of us looked for a job before. We also search training courses to equip ourselves for the job market. Some of us also try to find suitable candidates for their team or try to train up their staff. What challenges do these tasks have in common? With limited time to search, we have:

1. An explosive amount of choices
2. Choices that are mostly similar to each other
3. Data, while somewhat categorized, the differentiators are in words, i.e.: natural languages
4. Different preferences with different background – the best paid Java programmer position may be easy to search, but are you up to it?

A description of yourself, i.e.: a resume, is ultimately what gets you into an interview, after the broad-stroke categorizations like “Java programmer”. Keyword search engines like google are common, but what if you can search with your resume?

Similarity, a search with job description for the best candidate, or the relevancy of a training to a job we want would be nice.

Vec2rec is a recommendation engine that, with natural language processing, enable us to search with a description, be it a resume, a job description or a training description for most relevant results.

In addition, we can also run **what-if scenarios**, e.g.: with my current resume, how much closer would it be for me to a dream job if I take this training course?

# Use cases

* Job Seekers
  + Find the most suitable jobs
* Headhunters
  + Find the most suitable candidates for a job
* Training Professionals / Managers / Job Seekers
  + Most relevant trainings to a job
  + What jobs a training can enable
  + Increment of a candidate’s suitability for a job after a training

# Architecture



## Components

### Batch processing – to produce Gensim Doc2Vec models for similarity lookups

* There are 4 types of artifacts, raw data, preprocessed generic data, Gensim Doc2vec formatted training and testing data, and trained model(s). All of these are stored on S3.
* The Kubeflow Pipeline enables modularization and reruns. Each stage is a docker which can be replaced, even with non-python dockers as long as it produces results with correct format

#### Kubeflow Pipeline Phases

The following are all dockers images uploaded to DockerHub. The job definitions are written in Python which are compiled with the Kubeflow domain specific compiler (DSL) into a yaml file, which can be uploaded into the Kubeflow cluster for job definition. Each step can either pass “small” variables, usually int, str, float, bool. For larger data will have to serialize and Kubeflow will help to copy to the correct path location to be retrieved by the next phase.

* **Generic preprocessing phase** converts raw data, such as Excel and PDF into Pandas Dataframe, which is computed with Dask Dataframe / Dask Delayed, stored as Parquet
  + File format converter is implemented as descriptor to be easily replaceable
  + Preprocessing includes conversion to lower case, UTF-8 charset, removal of NLTK stopwords and punctuations, and the use of Krovetz Stemmer, which seems to have better conversion results than some others like the NLTK stemming or lemmatization; dropping of short and infrequent words
  + Preprocessing is implemented as a descriptor to be easily replaceable
* **Gensim Doc2vec preprocessing phase** split training/testing set in dataframes on Parquet and convert into Gensim Doc2Vec Training and Testing Corpus
  + Specifically detached from the generic phase to allow Doc2vec engine to be replaced
  + Final point where incremental updates are possible as Doc2vec models are not.
  + Currently merged w/ training phase as corpus serialization is not implemented.
* **Gensim Doc2vec training phase** builds the doc2vec model from training corpus
  + Currently, incremental update is not supported by Doc2Vec the model needs to be retrained whenever there are additions/removal to corpus
  + 4 models are built
    - Models from each data type – resumes, job desc, train desc
      * Better retrain performance
    - Model with all data meshed together
      * Larger sample size, more complete vocabulary
* **Gensim Doc2vec testing phase** uses the both the training data and testing data to evaluate the model performance. This phase is not exposed to the user.
  + Training data: should have best similarity to itself
  + Testing data: in this project, eyeball verification is employed though more sophisticated methods are available

### Front end – for Similarity Queries

* Includes
  + CLI Python Module with argparse
  + Flask API (if there is time)
* Functions equivalent to
  + def lookup(text, filepath, lookup\_type, topn=3, model=”all”, return\_type=”text”)
    - lookup with text or filepath
      * return\_type = “text”: returns a dict with top N {file text: similarity score}
      * return\_type = “path”: returns a dict w/top N {S3 URL: similarity score}
    - lookup\_type: type of data returned, valid values are [“resume”, “job”, “train”]
    - model: model used for lookup, valid values are [“all”, “single”]
  + def add\_doc(dir, file\_glob)
    - check doc validity and put into file repo for future model training
  + def del\_doc(dir, file\_glob)
    - remove file path for future model training

# Package Structure

## vec2rec.preprocess.tools

class TokenData: # only highlights are shown here

def \_\_init\_\_(self, chunksize=20):

... # set Dask chunksize

# descriptors w/ dask.delayed

extract\_pdf\_text = staticmethod(dask.delayed(PDFReader()))

tokenize = staticmethod(dask.delayed(Tokenizer()))

def pdf\_to\_df(self, parent\_dir, file\_glob="\*.pdf", df\_type="resume"):

... # with Dask, convert pdf to df, add to class var train/test df

def xls\_to\_df(self, parent\_dir, file\_glob="\*.xlsx", df\_type="train"):

... # with Dask, convert xls to df, add to class variable train/test df

def read\_parquet(self, parent\_dir, file\_path=default\_fp, df\_type="all"):

... # read parquet from local/S3 to df and add to class var train/test df

def to\_parquet(self, parent\_dir, file\_path=default\_fp, df\_type="all"):

... # write class variable train/test df to parquet on local/S3

# class PDFReader is a PDF2Text descriptor

class PDFReader: # only highlights are shown here

@staticmethod

def extract\_pdf\_text(path, fmt="string"):

...

# class Tokenizer is a descriptor that perform tokenizing

# stemming, and various data cleaning tasks from the doc

class Tokenizer: # only highlights are shown here

@staticmethod

def tokenize(text):

...

## vec2rec.frontend.vec2vec

class Vec2Rec: # the class used by the front end

# each of these models has a lookup() function which will download the required

# model from S3 to perform the similarity check if not already downloaded

# These are current Gensim models but can be others, see class NLPModel

job\_model = D2VModel()

res\_model = D2VModel()

train\_model = D2VModel()

all\_model = D2VModel()

def add\_doc(self, parent\_dir, file\_glob):

... # upload doc to S3 repository

def del\_doc(self, parent\_dir, file\_glob):

... # delete doc from S3 repository

## vec2rec.models.nlpmodels

class NLPModel:

# Abstract class with all functions implemented in D2VModel

class D2VModel(NLPModel):

def \_\_init\_\_(self, vector\_size=75, min\_count=2, epochs=40, test\_ratio=0.3):

... # Initialize model in class var

def build\_corpus(self, parent\_dir, file\_path):

... # reads in a parquet file from local/S3 and build corpus

def train(self, epochs=None):

... # builds vocab and train model in class var

def test(self, topn=3):

... # calculate accuracy with training data – doc itself should have

# highest similarity. The print out the topn similarity with the

# testing data

def load\_model(self, parent\_dir, file\_path):

... # load saved model from local/S3 & store in class var

def save\_model(self, parent\_dir, filepath):

... # save model into file and upload to local/S3

def lookup(self, text, filepath, top\_n=3, return\_type="text"):

# lookup with text or filepath local or S3. Filepath can be a list

# return\_type = “text”: returns a dict with top N {file text: sim score}

# return\_type = “path”: returns a dict w/top N {S3 URL: similarity score}

# lookup\_type: data type returned in [“resume”, “job”, “train”]

# model: model used for lookup, valid values are [“all”, “single”]

## vec2rec.kfp.pipeline

Functions in this file is used to generate the definition file in yaml.

Each step returns a dsl.ContainerOp object which will ultimately be a runnable docker container in the pipeline.

The decorator @dsl.pipeline is used to define the pipeline where results are passed to the next phase. Care must be taken to remember these are not real python functions but dockers. All variables passed between the phases needs to be small (< 256k), or be serialized into files. It could either be remote storage like S3, or passing locally. Using the options file\_outputs and artifact\_argument\_path, Kubeflow will help you to copy the files from one container to another.

Each phases will need to be built as a docker and uploaded into docker hub or GCR and downloaded as the pipeline runs.

This kind of structure forces you to restructure your code and can be very cumbersome as all variables are not shared.

Python functions can also be directly converted into container phases without uploading containers, tho these are still functions which does not share variables with other steps.

import kfp

from kfp import dsl

def preprocess\_op(parent\_dir=S3\_BUCKET\_BASE, chunk=20, doc\_type="all", local\_dir=LOCAL\_BASE):

...

return dsl.ContainerOp(

name="preprocess",

image=f"crispinnosidam/vec2rec:{img\_ver}",

command=["pipenv", "run", "python", "-W", "ignore", "-m", "vec2rec",],

# ...

file\_outputs=[...], # file path to be passed to next step

)

def train\_op( resume\_path, job\_path, train\_path, parent\_dir=S3\_BUCKET\_BASE,

vector\_size=75, min\_cnt=2, epochs=100, test\_ratio=1 / 3, doc\_type="all", local\_dir=LOCAL\_BASE, ):

...

return dsl.ContainerOp(

name="train",

image=f"crispinnosidam/vec2rec:{img\_ver}",

artifact\_argument\_paths=[

dsl.InputArgumentPath(argument=resume\_path),

dsl.InputArgumentPath(argument=job\_path),

dsl.InputArgumentPath(argument=train\_path),

], # file paths to this step

command=["pipenv", "run", "python", "-W", "ignore", "-m", "vec2rec",]

+ ["-p", parent\_dir, "train", ],

# ... other params

file\_outputs=[...], # file paths to be passed to next step

)

def test\_op(input\_path, parent\_dir=LOCAL\_BASE, sample=1, top\_n=2, doc\_type="all"):

return dsl.ContainerOp(

name="test",

image=f"crispinnosidam/vec2rec:{img\_ver}",

artifact\_argument\_paths=[...], # file paths to this step

command=["pipenv", "run", "python", "-W", "ignore", "-m", "vec2rec",]

+ ["-p", parent\_dir, "test", ]

# ... other params

)

@dsl.pipeline(name="my testing pipeline", description="my testing pipeline description")

def vec2rec\_pipeline(AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY):

...

pp\_op = preprocess\_op(S3\_BUCKET\_BASE)

tr\_op = train\_op(

pp\_op.outputs["resume"], pp\_op.outputs["job"], pp\_op.outputs["train"]

)

tt\_op = test\_op(tr\_op.outputs)

...

if \_\_name\_\_ == "\_\_main\_\_":

kfp.compiler.Compiler().compile(vec2rec\_pipeline, \_\_file\_\_ + ".yaml") # compile defn to yaml

# Technology Stack

* Gensim Doc2vec
* Python – Descriptors, Iterators
* Kubflow Pipeline and Docker
* Dask Dataframe, Dask Delay, Pandas and Parquet
* Natural Language toolkit (NLTK) and Krovetz Stemmer, PyPDF4
* S3
* Flask

# Design Choices & Implementation Considerations

## 1) Preserve computation steps to allow easy reruns under different config / data updates

* Kubeflow pipeline allows easy reruns for each step.
* The serialization of the 3 generated artifact types minimize the number of reruns needed
  + For activities such as
    - Addition/Remove of data
    - Retrain with different parameters
    - Restarts
  + Features
    - Generic processed data can be updated incrementally
    - Doc2Vec formatted data cannot be updated incrementally but left provision for future enhancement for serialization method such as pickle
    - Segregated model avoid total retrain for doc types on changes
    - Saved Models avoid total recalculation of models over restarts

## 2) Modularize components and enable future enhancement / replacement

* Docker phases in Kubeflow allow replacement for whole phases
* Usage of Descriptor in PDF scrapper, Stemmer, data cleaning modules, even the main engine Doc2vec, allow the easy replacement of these modules

## 3) Enhance parallelization on computation and Memory Efficiency

* Use of Dask Dataframe and Dask Delayed increases parallelization
* Some attempts are made to reduce memory footprint during preprocessing by using iterators, but Doc2vec requires whole corpus to be in memory during training
* Temporary training data in Doc2vec model is deleted after training to reduce memory full print for lookup engine

## 4) Allow incremental growth to lookup database

* Interface is added to modify raw data of lookup database
* Allow Incremental updates up to generic preprocessed data
* Gensim Doc2vec does not allow incremental update of models

## 5) Centralized Repository

* S3 being the most easily implemented repository for document-type raw data
* A database may have better performance as interim data storage, but still cannot store saved models

# Future Enhancements

* Weighted importance based on
  + record age – with decaying importance
  + user preferences based on past search and selections
  + add user profile for matching, if available
* Migration of S3 to database for better retrieval performance
* Visualization with Roadmaps with Resume being a starting point and jobs being goals (can be multiple), with each training being steps in the middle
* Serialization of Gensim Training / Testing Corpus with pickle to reduce repeated computation after adding samples.
* Another training module which allows incremental sample addition without total retrain.
* Support conversion from MS Word for resume.
* Front end replaced by a chatbot

# References

Doc2Vec <https://radimrehurek.com/gensim/auto_examples/tutorials/run_doc2vec_lee.html>

Kubeflow <https://www.kubeflow.org/>

Dask DataFrame <https://docs.dask.org/en/latest/dataframe.html>

Dask Delayed <https://docs.dask.org/en/latest/delayed.html>

Pandas <https://pandas.pydata.org/>

NLTK <https://www.nltk.org/>

Krovetz Stemmer <https://sourceforge.net/p/lemur/wiki/KrovetzStemmer/>

PyPDF4 <https://github.com/claird/PyPDF4>

AWS S3 <https://aws.amazon.com/s3/>

Flask <https://flask.palletsprojects.com/en/1.1.x/>

Training Data - Hotchkiss <https://www.hotchkiss.org/uploaded/documents/Academics/2017-18Hotchkiss_CourseCatalog.pdf>

Job Data - Nevada Pay Survey <https://www.nevadaemployers.org/wp-content/uploads/Updated-Job-Descriptions-2018.pdf>

Resume Data <https://github.com/JAIJANYANI/Automated-Resume-Screening-System>