Urbanwood Document Classifier

BBE Department

Oct-Dec 2019

Table of Contents

Environment Setup / Deployment

Clone the codebase from this repo (https://github.umn.edu/agarw228/document_nlp).

- Create a virtualenv using the following commands:
  - Need python 3.6.5_1 and java8 (use: `sudo yum install python36 python36-pip`) to install python3
  - Install pip3 and virtualenv using following commands (this worked in ec2):
    - `curl -O https://bootstrap.pypa.io/get-pip.py`
    - `sudo python3 get-pip.py`
    - `pip3 install virtualenv --user`
  - Create virtualenv: `python3 -m virtualenv AI_env`
  - Activate virtualenv: `source AI_env/bin/activate`
- Install dependencies by changing directories to: *document_nlp* and use command: `pip install -r requirements.txt`, In case there are issues:
  - For ` OSError: mysql_config not found`, use `sudo yum install -y mysql-devel`,
  - For `gcc` issue: `sudo yum install gcc`
  - For `Python.h` issue: `sudo yum install python36-devel`
  - and then after resolving issues run again: `pip install -r requirements.txt` to see if there are any more issues.
- After that, run: `python -m spacy download en`
- Install other dependencies using: `python` > `import nltk` > `nltk.download('')` *[stopwords, punkt, brown]* (Download the last three one by one in nltk.download)
- **Above instructions are for Linux, for ubuntu follow these instructions. After these instructions run:** `pip install -r requirements.txt`
  - `sudo apt update`
  - `sudo apt get python3-pip` (After this install virtualenv and follow rest of the steps)
  - `sudo apt-get install libmysqlclient-dev`
  - `sudo apt-get install build-essential`
  - `sudo apt-get install libssl-dev`

- **Everything in the rest of the document would happen in this active virtual environment.**

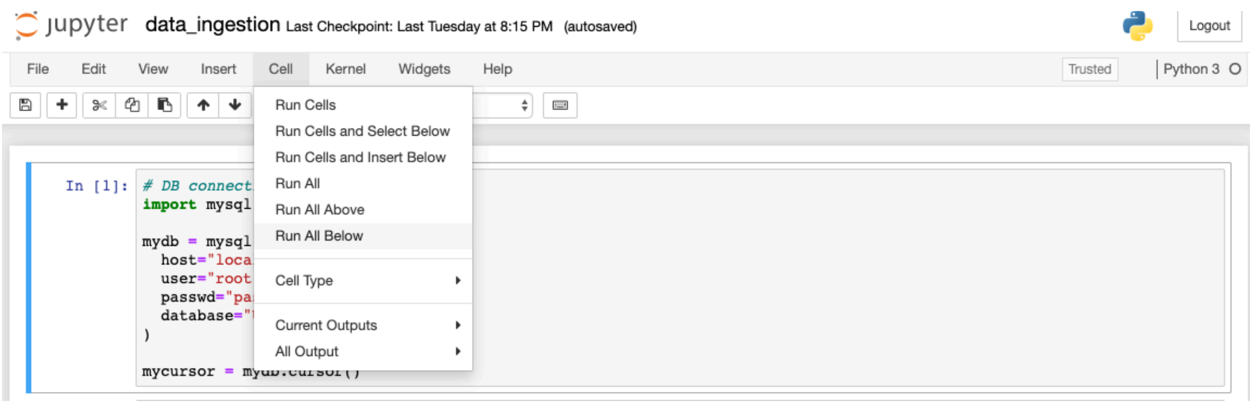
Running Server

- When above environment is setup and activated, go inside folder `Docproc` and run command:
  - `python manage.py migrate`
  - `python manage.py runserver 0.0.0.0:8000`
- This would activate the server with microservices.
- To check whether server is running or not hit `*local_test*` service provided with the postman collection.

Note: In this entire document, wherever details of connection mentioned such as (IP address, hostname, username, passwords, database names etc.) change them as per your requirements.

Data Ingestion

- Data ingestion is an offline + online process (offline component to read/parse pdf files and scrape links and online component to provide API interface for database ingestion)
- Create `training_data` table, after getting into **swadm,** using the following commands:
    - `mysql -h db_host -u username -p db_name`
    - `` CREATE TABLE `training_data` (`document_name` varchar(200) NOT NULL,`document_text` mediumtext NOT NULL,`category` varchar(100) NOT NULL); ``
- Follow these steps to load documents from folders containing pdf files (subfolders determining the category) or from files containing web links:
    - **Make sure in training data that each document belongs to only one subfolder** i.e. there are no repeated categories, one category only for each document.
    - Place this folder inside `document_nlp/DocProc/data_ingestion/data/`
    - Change directory in the terminal to `data_ingestion` and run command: `jupyter notebook`
    - The above command would open an interface in a web browser, from this interface open the file **data_ingestion.ipynb.** This is the file to perform data ingestion.
    - Change details in the first *constants cell* to where data needs to be ingested and then run the cells one by one. Each cell has description about its functionality.

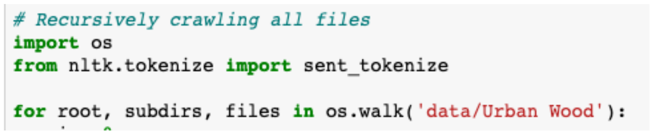

jupyter data_ingestion Last Checkpoint: Last Tuesday at 8:15 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run Cells
Run Cells and Select Below
Run Cells and Insert Below
Run All
Run All Above
Run All Below
Cell Type
Current Outputs
All Output

```
In [1]: # DB connect
        import mysql

        mydb = mysql
          host="loca
          user="root
          passwd="pa
          database="
        )

        mycursor = mydb.cursor()
```

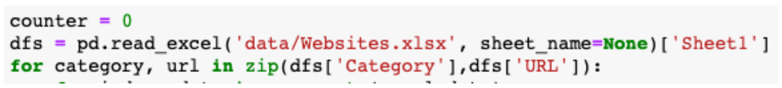
- Run next three cells (cell #2: module to create http request payload for online data ingestion request, cell #3: preprocess text data, cell #4: module to parse pdf files)
- In cell #5, mention relative path of the folder containing the training documents and run this cell. This will recursively read pdf files and populate DB via data_ingestion srvc.



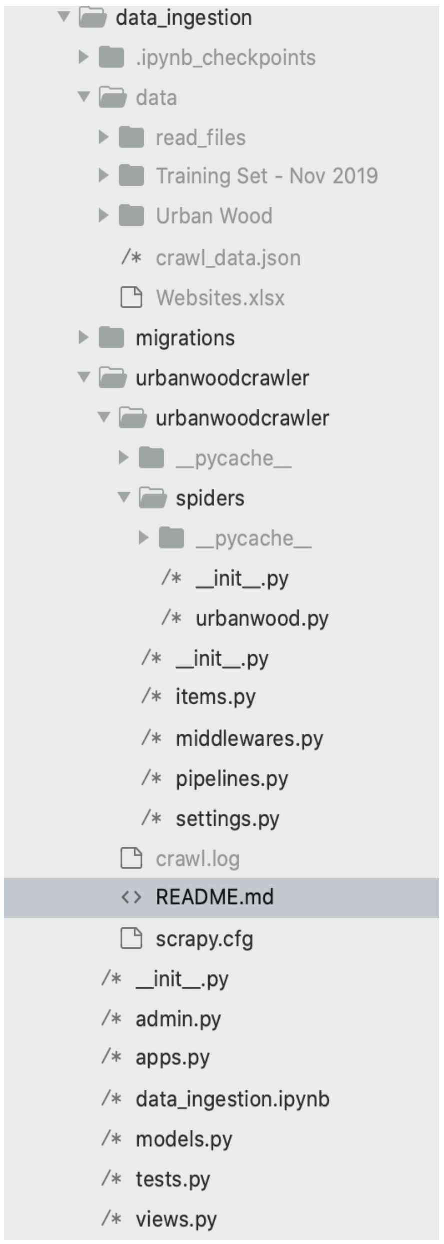
```
# Recursively crawling all files
import os
from nltk.tokenize import sent_tokenize

for root, subdirs, files in os.walk('data/Urban Wood'):
```

- In order to store documents, form web links into table, run cell #6, but mention the relative path of links file **(**has to be **.xlsx** with headers **Category, URL)**, but prior to this we would have to scrape data by running the scraper **urbanwoodcrawler.**



```
counter = 0
dfs = pd.read_excel('data/Websites.xlsx', sheet_name=None)['Sheet1']
for category, url in zip(dfs['Category'],dfs['URL']):
```



```
▼ data_ingestion
    ▶ .ipynb_checkpoints
    ▼ data
        ▶ read_files
        ▶ Training Set - Nov 2019
        ▶ Urban Wood
          crawl_data.json
          Websites.xlsx
    ▶ migrations
    ▼ urbanwoodcrawler
        ▼ urbanwoodcrawler
            ▶ __pycache__
            ▼ spiders
                ▶ __pycache__
                  __init__.py
                  urbanwood.py
              __init__.py
              items.py
              middlewares.py
              pipelines.py
              settings.py
          crawl.log
          README.md
          scrapy.cfg
      __init__.py
      admin.py
      apps.py
      data_ingestion.ipynb
      models.py
      tests.py
      views.py
```

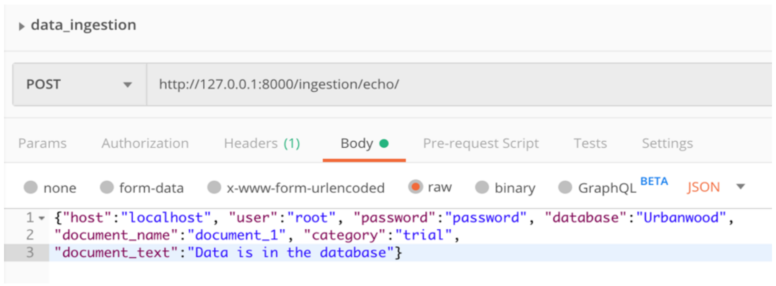
- Place the Websites.xlsx in the specified path inside data folder of data_ingestion module.
- Read README.md inside data_ingestion to get instructions on how to run the crawler.
- The crawler would read links from Websites.xlsx and it will create crawl_data.json, that contains text of all the crawled links.
- This document then would be used in the later cells to simply store scraped data from links into the Database.

Note: Always prefer parsing of .pdf weblinks instead of scraping them from internet as the scraper does a bad job at scraping pdf files from web. Thus, for such links best approach is to download them and save them along with other pdfs to be read by the parser.

Note: While inserting new training data, also prefer to do a complete data insertion rather than data updation. Thus, before complete data insertion clear already existing data using:

- `delete from training_data;`

-

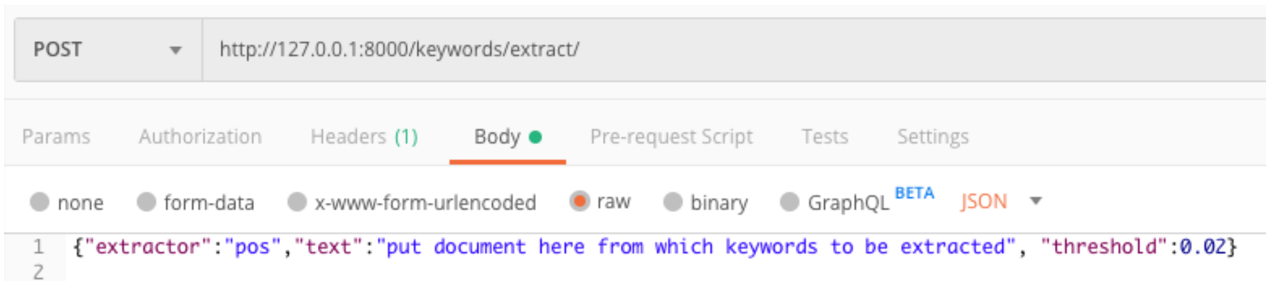


http://127.0.0.1:8000/ingestion/echo/

```
{"host":"localhost", "user":"root", "password":"password", "database":"Urbanwood",
"document_name":"document_1", "category":"trial",
"document_text":"Data is in the database"}
```

Payload of data ingestion service (online component)

Keywords extraction



http://127.0.0.1:8000/keywords/extract/

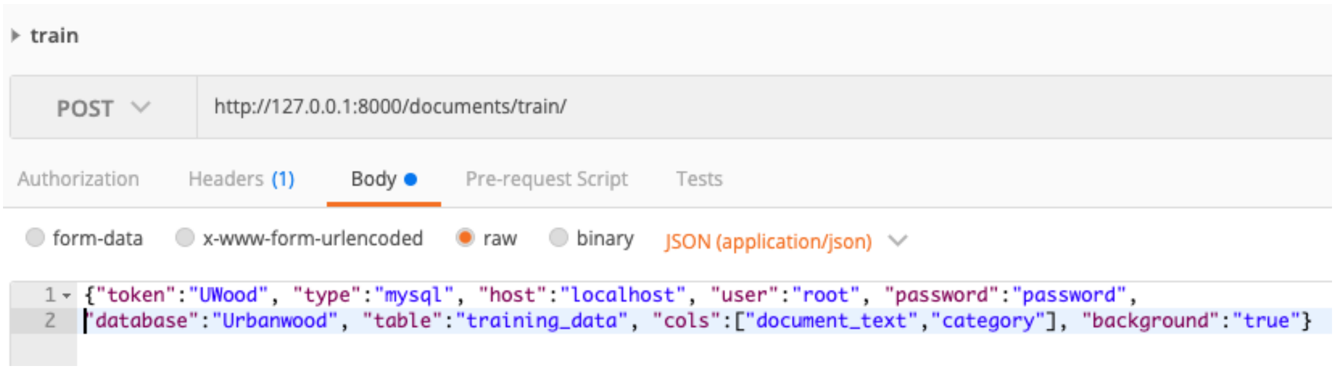
```
{"extractor":"pos","text":"put document here from which keywords to be extracted", "threshold":0.02}
```

- **extractor**: can be ***pos*** *or* ***rake***. The former is based on part-of-speech tagging an extracts significant nouns and n-grams (both can be configured in pos.py file of `keywords` module). The latter is based on `rake` algorithm of keyword extraction and the parameters of extraction can be configured in rake.py and rake_algo.py files of `keywords` module.
- **text**: send document from which keywords to be extracted from. Since it needs to be sent as a string, make sure it does not contain (\n, " and \t) characters while testing the service with Postman, as it leads to bad parsing of json.
- **threshold**: works only for type pos (for rake all configuration would have to be made via code). It indicates the top percentage of keywords to be returned back as relevant keywords (e.g. if system found 1000 keywords, threshold 0.02 will return 20 most significant keywords).

Note: keyword extraction is a task that has no perfect solution. It is subjective and differs from algorithm to algorithm. For maximum information, can even combine keywords obtained from both `rake` and `pos` based extraction methods.

Training

- After multiple iteration, with the currently available data the model that works best is: Basic language preprocessing + mean word embedding vectorizer + Logistic regression
- To change this model, one would have to tweak the `**background_train**` module within the ***documents/main.py*** file.
- This is a generic service and model of these kind can be prepared in multiple numbers from multiple database service using the following REST service.



```
{"token":"UWood", "type":"mysql", "host":"localhost", "user":"root", "password":"password",
"database":"Urbanwood", "table":"training_data", "cols":["document_text","category"], "background":"true"}
```

- ***token***: unique identifier as we can prepare multiple models for multiple use cases (e.g. UWood for urban wood documents, UWood-bin for urban wood binary classifier, DWood for dry wood documents etc.)
- ***type***: mysql or file. The former means data from mysql database and latter means data from .csv file. In the latter case the csv file has to be places at a specific location i.e. (`*document_nlp/DocProc/documents/static/documents/*.csv*`*).* In case of file, the request is:

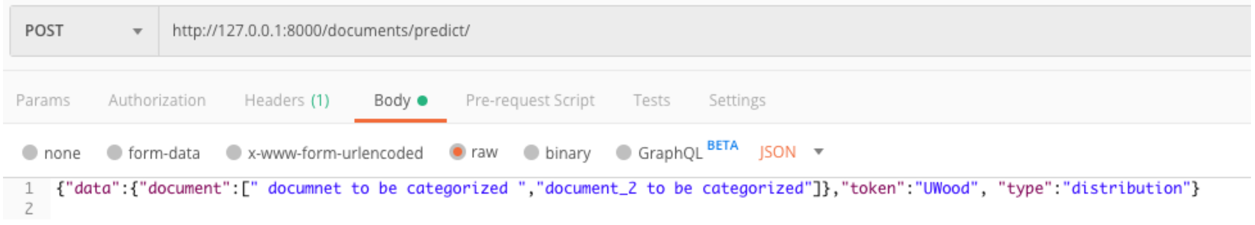
```
{"token":"UWood", "type":"file", "filename":"BBC News Train.csv"}
```

Below fields only relevant when type is `*mysql*`

- ***host***: hostname where database is hosted
- ***user***: username to login to database
- ***password***: password to login to database
- ***database***: database name in which data table resides
- ***table***: table on which data resides (can change this to train on new data)
- ***cols***: array of column names in table which have document and their categories
- ***background***: true/false. As training might take time, the user request cannot be made to wait. With `*true*`, the training can be made to run in the background and user request is returned back *(asynchronous training)*. If training is to be made synchronous, don't include this field or make it `*false*`.

Note: In order to retrain the system, either clear the table `training_data` or train from a new table.

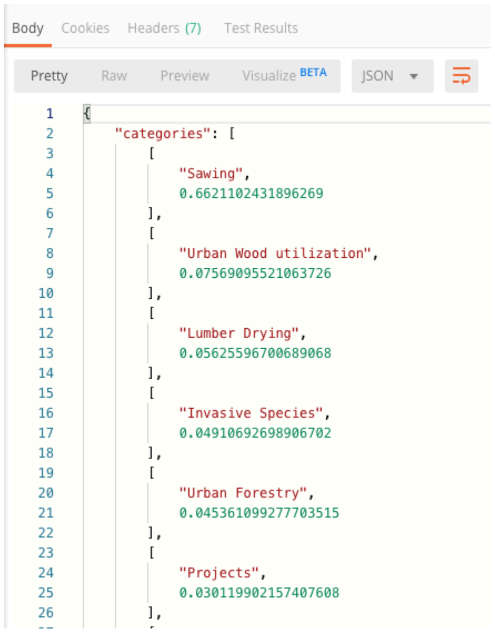
Prediction



POST http://127.0.0.1:8000/documents/predict/

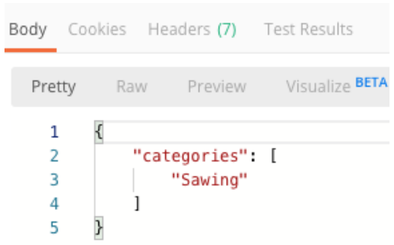
```
{"data":{"document":[" documnet to be categorized ","document_2 to be categorized"]},"token":"UWood", "type":"distribution"}
```

- The most important field is the **token**. Since there might exist multiple models, created by train service using multiple tokens, make sure the right one is selected for prediction. (for e.g. if need to predict over Urban wood data trained using token *UWood*, make sure that specific token is selected with the predict service.
  - **Note**: tokens are case-sensitive
- Can send either one document for predicting category or list of documents (bulk prediction) in single request using field ***{data : {document: [****comma separated list of documents* ***]}}*.** Each document has to be a string (thus, make sure that no (\n, " and \t) are present in the document when using Postman client to paste any document and call predict service.
- **type:** can be *distribution* or *single*. If field not included, then by default behavior is single. `distribution` returns the results as a probability distribution of document among all categories. `single` or default behavior just returns category with the highest probability.



```
{
    "categories": [
        [
            "Sawing",
            0.6621102431896269
        ],
        [
            "Urban Wood utilization",
            0.07569095521063726
        ],
        [
            "Lumber Drying",
            0.05625596700689068
        ],
        [
            "Invasive Species",
            0.04910692698906702
        ],
        [
            "Urban Forestry",
            0.045361099277703515
        ],
        [
            "Projects",
            0.030119902157407608
        ],
```

distribution

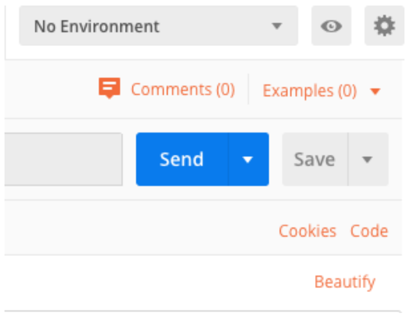


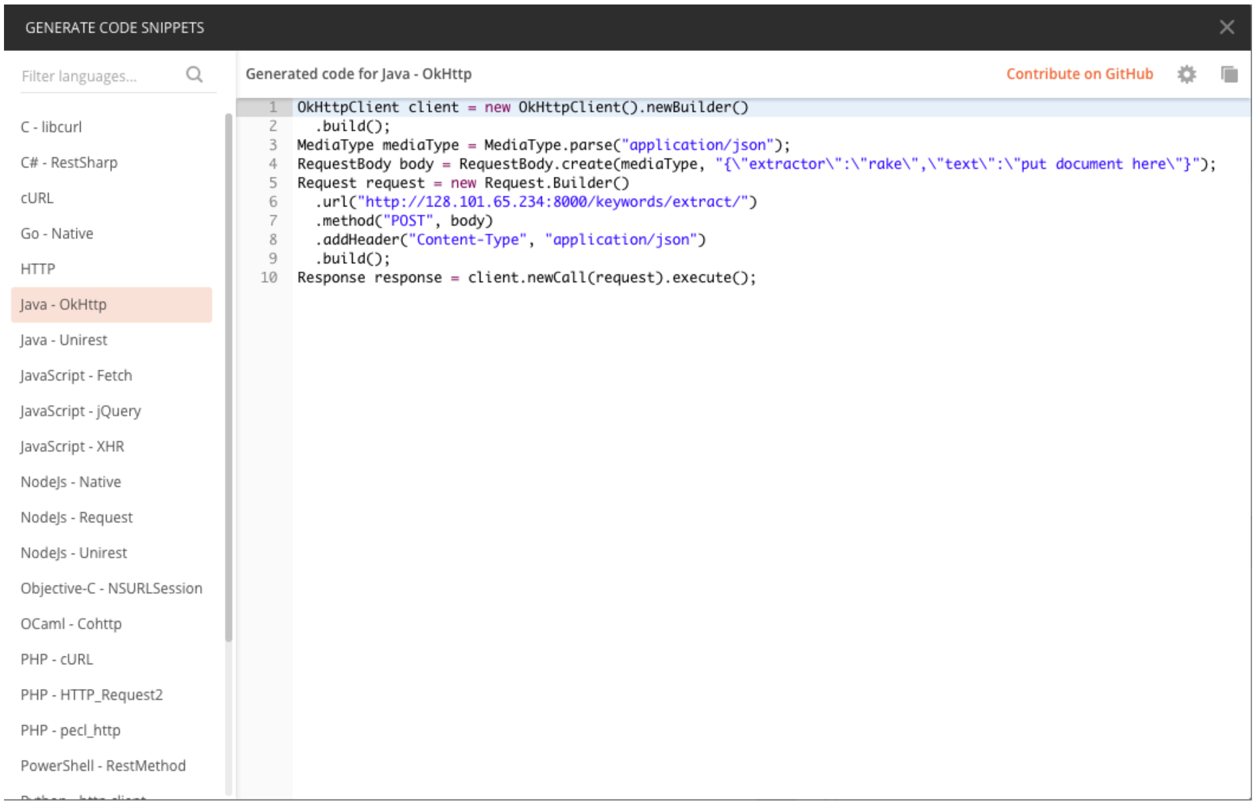
```
{
    "categories": [
        "Sawing"
    ]
}
```

single

Integration with the services

- To integrate these RESTful services with any programming language, what can be done is load the postman collection into postman and see the **Code** for each service in required programming language. Refer to picture below for more context.





Generated code for Java - OkHttp

```java
OkHttpClient client = new OkHttpClient().newBuilder()
  .build();
MediaType mediaType = MediaType.parse("application/json");
RequestBody body = RequestBody.create(mediaType, "{\"extractor\":\"rake\",\"text\":\"put document here\"}");
Request request = new Request.Builder()
  .url("http://128.101.65.234:8000/keywords/extract/")
  .method("POST", body)
  .addHeader("Content-Type", "application/json")
  .build();
Response response = client.newCall(request).execute();
```

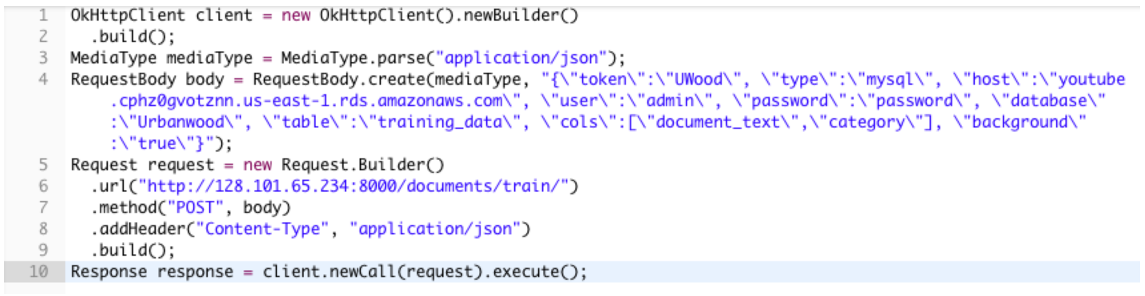
extract service

- For Java the code for services is given below:
    - extract (for keyword extraction):

```
OkHttpClient client = new OkHttpClient().newBuilder()
  .build();
MediaType mediaType = MediaType.parse("application/json");
RequestBody body = RequestBody.create(mediaType,
"{\"extractor\":\"rake\",\"text\":\"put document here\"}");
Request request = new Request.Builder()
  .url("http://128.101.65.234:8000/keywords/extract/")
  .method("POST", body)
  .addHeader("Content-Type", "application/json")
  .build();
Response response = client.newCall(request).execute();
```

    - train (to train model by taking documents from provided database):

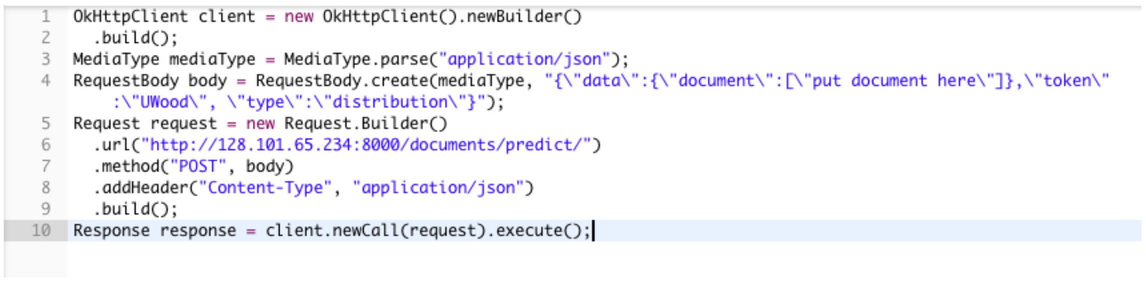
```
OkHttpClient client = new OkHttpClient().newBuilder()
  .build();
MediaType mediaType = MediaType.parse("application/json");
RequestBody body = RequestBody.create(mediaType, "{\"token\":\"UWood\",
\"type\":\"mysql\", \"host\":\"youtube.cphz0gvotznn.us-east-
1.rds.amazonaws.com\", \"user\":\"admin\", \"password\":\"password\",
\"database\":\"Urbanwood\", \"table\":\"training_data\",
\"cols\":[\"document_text\",\"category\"], \"background\":\"true\"}");
Request request = new Request.Builder()
  .url("http://128.101.65.234:8000/documents/train/")
  .method("POST", body)
  .addHeader("Content-Type", "application/json")
  .build();
Response response = client.newCall(request).execute();
```



```
OkHttpClient client = new OkHttpClient().newBuilder()
  .build();
MediaType mediaType = MediaType.parse("application/json");
RequestBody body = RequestBody.create(mediaType, "{\"token\":\"UWood\", \"type\":\"mysql\", \"host\":\"youtube
    .cphz0gvotznn.us-east-1.rds.amazonaws.com\", \"user\":\"admin\", \"password\":\"password\", \"database\"
    :\"Urbanwood\", \"table\":\"training_data\", \"cols\":[\"document_text\",\"category\"], \"background\"
    :\"true\"}");
Request request = new Request.Builder()
  .url("http://128.101.65.234:8000/documents/train/")
  .method("POST", body)
  .addHeader("Content-Type", "application/json")
  .build();
Response response = client.newCall(request).execute();
```

train service

- predict (to predict category of document using token to load pretrained model):
  -

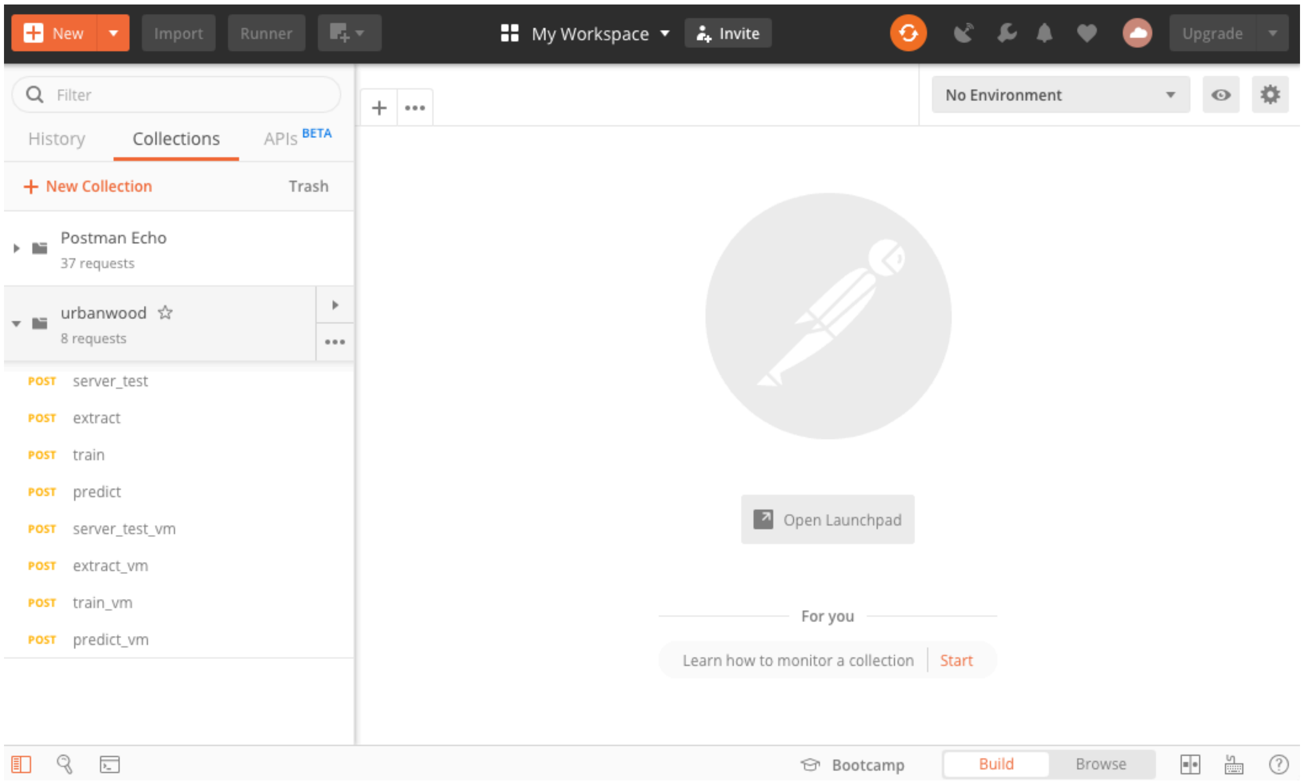


```
OkHttpClient client = new OkHttpClient().newBuilder()
  .build();
MediaType mediaType = MediaType.parse("application/json");
RequestBody body = RequestBody.create(mediaType, "{\"data\":{\"document\":[\"put document here\"]},\"token\"
    :\"UWood\", \"type\":\"distribution\"}");
Request request = new Request.Builder()
  .url("http://128.101.65.234:8000/documents/predict/")
  .method("POST", body)
  .addHeader("Content-Type", "application/json")
  .build();
Response response = client.newCall(request).execute();
```

predict service

```
OkHttpClient client = new OkHttpClient().newBuilder()
  .build();
MediaType mediaType = MediaType.parse("application/json");
RequestBody body = RequestBody.create(mediaType, "{\"data\":{\"document\":[\"put
document here\"]},\"token\":\"UWood\", \"type\":\"distribution\"}");
Request request = new Request.Builder()
  .url("http://128.101.65.234:8000/documents/predict/")
  .method("POST", body)
  .addHeader("Content-Type", "application/json")
  .build();
Response response = client.newCall(request).execute();
```
Webservices collection (Postman client)

- Download postman application (an HTTP client software for testing HTTP APIs).
- Import the collection provided in the file (**urbanwood.postman_collection.json**):
  - Use the services extract, train and predict to test services while running them in local.
  - Use the services extract_vm, train_vm and predict_vm to interact with services deployed over the VM.
- The payload of the train service can be changed to load new dataset and prepare the model on new data.
- Make sure to have same tokens in the *train* & *predict* service, to get prediction on the same model trained.
- **Note**: Make sure while putting document in payload of *extract* & *predict* services to remove (\n, " and \t) as these prevent the documents to be parsed as proper json. The documents need to be sent a single string and these characters prevent that.

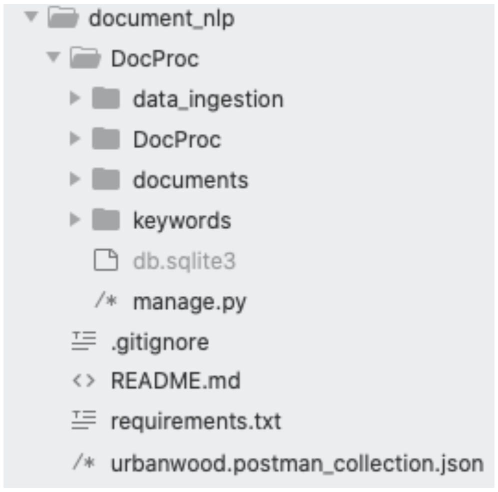


Postman Echo
37 requests

urbanwood
8 requests

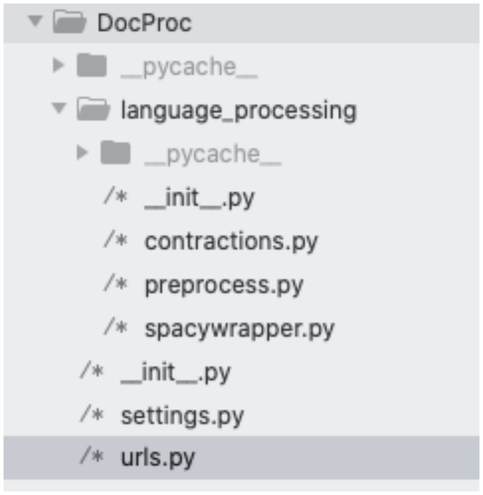
POST server_test
POST extract
POST train
POST predict
POST server_test_vm
POST extract_vm
POST train_vm
POST predict_vm

Codebase Overview



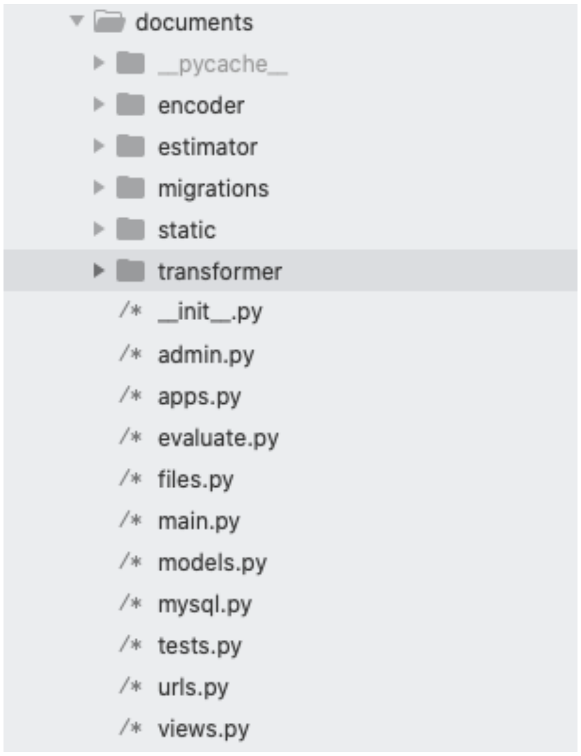
```
document_nlp
  DocProc
    data_ingestion
    DocProc
    documents
    keywords
    db.sqlite3
    manage.py
  .gitignore
  README.md
  requirements.txt
  urbanwood.postman_collection.json
```

- Three main modules are:
  - **data_ingestion:** contains code for reading pdfs and scraping text from web links to be loaded into the database.
  - **documents**: contains code for document classification (training and prediction)
  - **keywords**: contains code for keyword extraction from any document



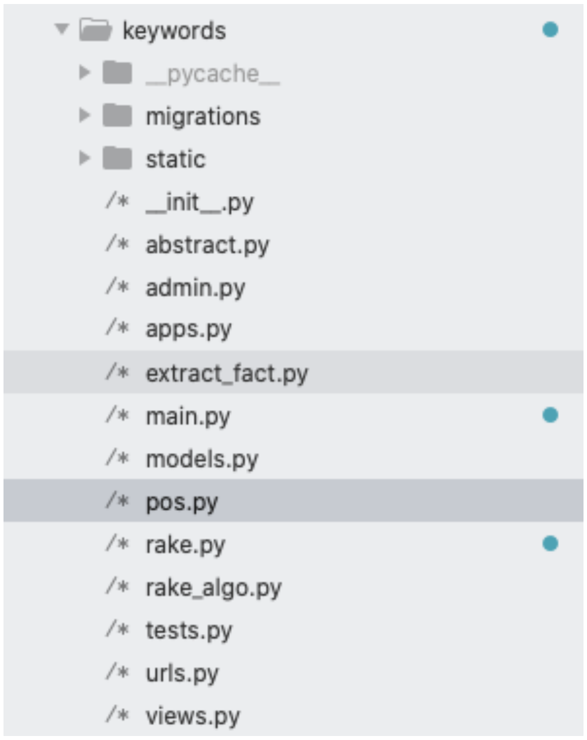
```
DocProc
  __pycache__
  language_processing
    __pycache__
    __init__.py
    contractions.py
    preprocess.py
    spacywrapper.py
  __init__.py
  settings.py
  urls.py
```

- DocProc just contains one module (**language_processing**) to cleanse and process natural text in the documents.

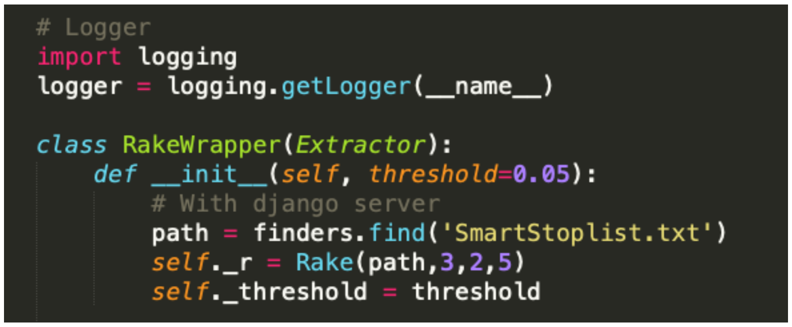


```
▼ documents
  ▶ __pycache__
  ▶ encoder
  ▶ estimator
  ▶ migrations
  ▶ static
  ▶ transformer
    __init__.py
    admin.py
    apps.py
    evaluate.py
    files.py
    main.py
    models.py
    mysql.py
    tests.py
    urls.py
    views.py
```

- `**documents**` module is the crux as it contains the entire classification pipeline:
  - **encoder** contains code to encode categorical data into labels and then store the model.
  - **transformer** contains code to transform text data into vectors. Contains different implementations and one can be chosen using factory class.
  - **estimator** uses machine learning algorithms to make predictor after data is cleansed and transformer. Contains many algorithms from which any could be chosen to test the data.
  - **evaluate.py** contains evaluation modules to provide validation accuracy on a chosen algorithm.
  - **files.py** is a generic module to store different types of files, including machine learning models in this case.
  - **main.py** contains runner code (client for training and prediction services). The control comes here from the view (request services) and entire processing happens within this module.



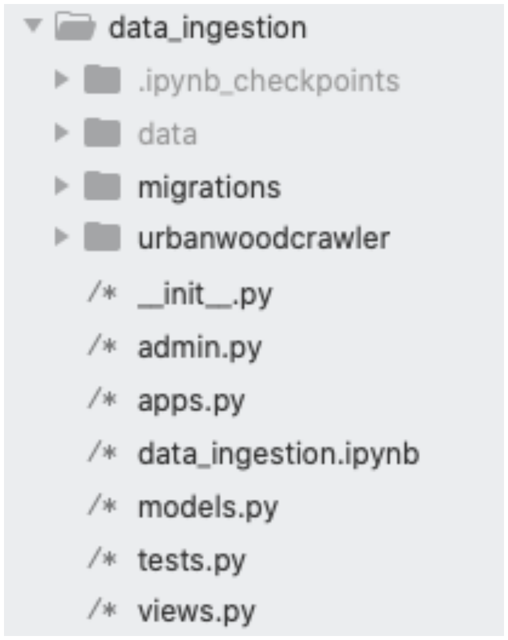
- `**keywords**` module for keyword extraction from any generic document. Need not require any training data as based on known algorithms and grammar parsing rules:
    - **pos.py** contains class that implements part-of-speech based keyword extraction.
    - **rake.py** contains class that implements rake-based keyword extraction.
    - **rake_algo.py** has open source code for the rake keyword extraction algorithm that we encapsulate in our class.
    - **main.py** contains runner code (client for extraction). The control comes here from the view (request services) and entire processing happens within this module.



```
# Logger
import logging
logger = logging.getLogger(__name__)

class RakeWrapper(Extractor):
    def __init__(self, threshold=0.05):
        # With django server
        path = finders.find('SmartStoplist.txt')
        self._r = Rake(path,3,2,5)
        self._threshold = threshold
```

The no. of keywords extracted for `pos` can be controlled at the service level, but for `rake`, changes have to be made in the above shown module.


```
data_ingestion
  .ipynb_checkpoints
  data
  migrations
  urbanwoodcrawler
  __init__.py
  admin.py
  apps.py
  data_ingestion.ipynb
  models.py
  tests.py
  views.py
```

- `**data_ingestion**` module used to quickly load document and category data from pdf files and web links:
  - **urbanwoodcrawler** contains code for python-based web-scraper
  - **data_ingestion.ipynb** is a jupyter notebook that contains entire code for data ingestion. Each cell in the notebook perform certain function (details are mentioned in the data ingestion section of the document). This code only runs both `Tika parser` to read pdf files and ` urbanwoodcrawler` web scraper to read text and store everything onto database to prepare the training data.

Deployment

- Running on Django server using nohup (to run in the background) [not recommended for production purposes] (close terminal after following below steps, logs store in file **nohup.out)**
  - The code is present in **/swadm/classifier/document_nlp** (it is a git repo)
  - `source AI_env/bin/activate`
  - `cd DocProc`
  - `nohup python manage.py runserver 0.0.0.0:8000`

- To see and kill processes running at port
  - `lsof -i :8000` (to get the PID)
  - `kill -9 PID`