SparkCognition DeepNLP ™
API User Guide

A SparkCognition ™ Education Document

Q3-2019
v2.1 - 7.2019

SparkCognition DeepNLP API User Guide

Contents

About this guide 3

Ingestion 3
- Ingestion and Data Warehousing . . . 3
  - Storage Design . . . 3
  - Data Layout . . . 4
- REST API (Ingestion) . . . 5
  - Document Collection . . . 5

Collections 5
- Create a new Collection . . . 5
- List Collections . . . 6
- Delete a Collection . . . 6
- Collection Details . . . 6

Cabinets 7
- Create a new Cabinet . . . 7
- List Cabinets . . . 8
- Delete a Cabinet . . . 8

Ingestion Requests 8
- New Ingestion Request . . . 8
- List Ingestion Requests . . . 9
- Ingestion Request Status . . . 9

Classification - Easy Way 10
- Train Cabinet . . . 10

Classification - Configurable Way 10
- Classifier Request . . . 11
- Preliminary Classifier Retrieval . . . 12
- Active Classifier Retrieval . . . 12
- Preliminary Classifier Status . . . 12
- Active Classifier Status . . . 12
  - Setting Current Active Classifier . . . 13
- Embeddings . . . 13
  - Request Word Embedding . . . 13

List Word Embeddings . . . 14
Check Word Embedding Status . . . 14
Embedding Parameters . . . 14
Embedding Request Parameters . . . 14
Pattern Parameters . . . 14
Composite Parameters . . . 14
Classification Parameters . . . 14
Classifier Request Parameters . . . 14
SVM Parameters . . . 15
Neural Network Parameters . . . 15

Query Documents **16**
Query . . . 16
Query With Keywords . . . 17
Table . . . 17
Entities . . . 18
Create New Entity Definition . . . 18
List Entity Definitions . . . 19
Create New Entity Occurrence . . . 19
List Entity Occurrences . . . 20
Modify an Entity Occurrence . . . 20
Delete an Entity Occurrence . . . 21
Analytics on Entity . . . 21

Transformations **22**
Definitions . . . 25
Rules for Transformation Functions: . . . 25
Transform . . . 26
Retrieve Available Transformations . . . 27
Entity Extraction . . . 28
Request Training . . . 28
Request Extraction . . . 28
Entity Extraction Status . . . 28
Pdf Endpoint . . . 29
getpdf endpoint . . . 29

Authentication **30**
Security Caveat . . . 30
Create User . . . 30
Login . . . 30
Create Role . . . 31
Add Role to User . . . 31
Get Roles of User . . . 31

Contact Support **32**

About this guide

This guide describes the Sparkcognition DeepNLP™ REST APIs. This guide is intended for data scientists, software engineers and analysts who want to interact with the DeepNLP system. Note that throughout this document, some long key and token values are truncated - indicated by ellipses (. . .) - or broken across lines, indicated with a backslash (\).

Ingestion

Ingestion and Data Warehousing

The DeepNLP system enables ingestion of small, frequent batches of documents while avoiding a *small-files* problem that is common with *Hadoop Distributed File Systems* (**HDFS**). The small-files problem arises with both HDFS and Yarn (Hadoop YARN) if they are configured with *namenode* and resource manager as standalone, non-distributed services. Eventually, and with a sufficient number of small files, one of the two services becomes overloaded. This overload results in the resource manager generating an `OutOfMemoryError` condition. This is more likely to occur when a submitted job requires computing splits.

Storage Design

The DeepNLP design divides data into two folder types:

- *archive* folders
- *current* folders

Note: SparkCognition DeepNLP requires that every input directory accessed by a job is partitioned or provisioned with these folders.

Archive folders hold a small number of merged files, usually for an extended period of time. In contrast, the *current* folders contain multiple (many) small, unmerged files, usually for short(er) periods of time. Both folders contain multiple subdirectories, each specific to a particular time frame or period.

Batch jobs read both archive and current data directories as input. Note that due to the timing of batch job processing it is possible for data to be processed more than once. This means issues can occur when a batch job is submitted concurrent to an *archive* folder being created from a *current* folder. The result is duplicate data that exists in both the *current* folder and target (copy) *archive* folder.

Solutions and Considerations

One solution to the dual data problem is to globally lock jobs, meaning that no jobs can run while data is being archived. This solution is suboptimal because it results in restrictions that impact performance. Various requirements must be met to implement this solution, including coordination between batch jobs and limiting job runtime, usually to less than 24 hours. The result is a general system slowdown.

A more flexible solution to this dual data problem is to create folders for each of the *current* and *archive* periods and adjust the behavior of the archiver to compensate, for example:

1. When the archiver runs, it signals all ingestion scripts to copy to a new subdirectory of *current*.
2. The archiver pauses until the ingestion scripts complete writing to the older *current* directory.

3. When the writing process completes, the archiver continues and merges the older (*current*) directory into its corresponding *archive* subdirectory.
4. Upon successful completion of the merge, the archiver deletes the earlier *current* directory.
   This means that at any point in time, batch jobs that require input data can safely access all current subdirectories in addition to those archive directories that do not have corresponding current directories.

Data formats and compression

Raw data file formats and file compression influence storage tactics. Storage must be managed to minimize system and performance impacts. Because uploaded files can consist of many small documents, storage is more efficient if the files are compressed. Currently, DeepNLP compresses the files as a group into an *Avro* file. Although raw files can be stored uncompressed, a best practice to manage storage space is to monitor space usage and define a space limit threshold. When that limiting threshold is breached, delete (or move) old raw files to recover storage space. See the Hadoop Avro documents for more information.

Data Layout

In terms of directory structure, each user requires a subdirectory on the HDFS system. The user directory name must follow the pattern: `/user/CODE_NAME_HERE/....`
Additionally, the DeepNLP ingestion and archival processes create specific directories on the HDFS system. The following table shows the created directories. Note that the format for `DATE` is *YYYY-mm-dd* and `TIMESTAMP` format is *YYYY-mm-dd-HH-MM*:

DeepNLP Required HDFS Directories

The following list details the purpose and location if the various required HDFS directories for DeepNLP:

For the following directory paths, the path shared by each is:
`/user/CODE_NAME_HERE/deepnlp/...`

Purpose	Location
raw data	`.../data/raw/TIMESTAMP_HERE`
archived squished data	`.../data/squished/archived/DATE_HERE`
current squished data	`.../data/squished/current/DATE_HERE`
archived split data	`.../data/split/archived/DATE_HERE`
current split data	`.../data/split/current/DATE_HERE`
archived parsed data	`.../data/parsed/archived/DATE_HERE`
current split data	`.../data/parsed/current/DATE_HERE`
archived label data	`.../data/labels/archived/DATE_HERE`
current label data	`.../data/labels/current/DATE_HERE`
Spark models	`.../models/spark/MODEL_NAME_HERE/TIMESTAMP_HERE`
Spark models	`.../models/tensorflow/MODEL_NAME_HERE/TIMESTAMP_HERE`

Purpose	Location
	• Format for `DATE` is *YYYY-mm-dd* • Format for `TIMESTAMP` is *YYYY-mm-dd-HH-MM*

Raw data is ingested using the DeepNLP `populate_hdfs` command in the `sparkcognition-platform` directory. Note the following division of responsibilities:

- *archiver* - archive, squish, split, and parse data
- *scheduler* - label data and train models

REST API (Ingestion)

Document Collection

This is a resource for document collections within DeepNLP that enables users to create new collections and view existing (stored) collections. Although each ingestion request is associated with a single URL, each collection can be associated with one or more ingestion requests. This means that each collection can be a combination of ingestions from different resources. Resources can be single documents or folders that contain documents and subfolders. Users can browse the document collection (by URL) to view files and folders associated with the collection.

DeepNLP API REST API Details DeepNLP's REST API allows the user to manually trigger ingestion, model training, classification, and updating Solr with the results. Use of JSON for Rest API All requests use JSON in order to encode messages that are part of the protocol, and the server requires two headers to be part of each request: Content-type: application/json Accept: application/json

Collections

A collection is a document corpus.

Create a new Collection

Route: `/collections/`

Method: POST

Body Parameters:

- *description*: arbitrary description of the data to ingest
- *name*: arbitrary name of the collection
- *url*: (optional) location of the data to ingest from. If this is not provided, an ingestion can be requested later for this collection. The URL must be in a format understood by nodes in the Yarn cluster to which the machine is submitted to

Example

```
curl \
  -XPOST \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \
  localhost:8080/collections/ \
  -d "{
    \"description\": \"description of collection\",
    \"name\": \"collection name here\",
    \"url\": \"hdfs://datastore.internal/data/my-raw-documents\"
  }"
```

List Collections

Route: `/collections/`

Method: GET

Example

```
curl -Haccept:application/json localhost:8080/collections/
```

Delete a Collection

Deleting a collection will also delete all of the cabinets contained within it.

Route: `/collections/COLLECTION_ID_HERE`

Method: DELETE

Example

```
curl \
  -XDELETE \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \
  localhost:8080/collections/1
```

Collection Details

Route: `/collection_details/`

Method: POST

Body Parameters

- *collection*: collection id of the collection
- *url* (optional): string that represents the absolute path of the directory/folder when empty root folders within the collection are returned

Request Parameter
cursorMark : needed for pagination (* for first call and needs to pass the returned *currorMark* for subsequent calls till the back end returns empty response)

Examples

```
1. curl -XPOST -s localhost:8080/collection_details/ -d '{"collection":"96"}'


2. curl-XPOST -s localhost:8080/collection_details/ -d '{"collection":"96",\
"url":"gs://deepnlp-sales=demo/"}'


3. curl -XPOST -s localhost:8080/collection_details/?cursorMark=* -d '\
{"collection":"16","gs://deepnlp-sales=demo/"}'


4. curl -XPOST -s localhost:8080/collection_details/?cursorMark=AoE/CTQ0MzUyN\
GRiYTY4ZDA0NjAzYzBmYmE4YTczZmQ5YTliODNjODVkOWI= -d '{"collection":"16","url":\
"gs://deepnlp-sales=demo/"}'
```

Cabinets

Note: Cabinets are not currently supported as a document classification mechanism.

Cabinets represent a user-defined view of a collection. Every cabinet is a set of folders and classifiers used for the same document collection. Different cabinets can be used, for example, by different business business units to classify the same dataset in many different ways.

Create a new Cabinet

Route: `/cabinets/`

Method: POST

Body Parameters:

- *cabinet_id*: id of the collection to which this cabinet belongs
- *name*: arbitrary name of the collection

Example

```
curl \
  -XPOST \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \
  localhost:8080/cabinets/ \
  -d "{
    \"name\": \"collection name here\",
```

```
    \"collection_id\": 1
  }"
```

List Cabinets

Route: `/cabinets/`

Method: GET

Example

```
curl -HAccept:application/json localhost:8080/cabinets/
```

Delete a Cabinet

Deleting a cabinet will delete it and all of the folders within it

.

Route: `/cabinets/COLLECTION_ID_HERE`

Method: DELETE

Example

```
curl \
  -XDELETE \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \
  localhost:8080/cabinets/1
```

Ingestion Requests

Ingestion requests are performed to ingest new documents. The `url` parameter is the most important: it points DeepNLP to the document location. The location can exist at any level within the location pointed to by the URL. The URL must represent a location readable by and accessible to the Yarn cluster being used by DeepNLP. All supported documents, including PDF documents, are ingested.

An ingestion is a discrete process where a set of documents are uploaded all at once and processed by DeepNLP. Every ingestion request is connected to a collection. Many ingestion requests can be made and processed for each collection.

New Ingestion Request

Route: `/ingestionrequests/`

Method: POST

Body Parameters:

- *description*: arbitrary description of the data to ingest
- *url*: Hadoop-interpretable, DeepNLP-accessible URL used to pull the data down
- *collection_id*: unique identifier of the collection where the data should be ingested

Example

```
curl \
  -XPOST \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \


   localhost:8080/ingestionrequests/ -d "{
     \"description\": \"docs\",
     \"url\": \"hdfs://datastore.internal/data/my-raw-documents-batch-2\",
     \"collection_id\": 1
   }"
```

List Ingestion Requests

This endpoint lists the status of every ingestion request (made so far) and their ID numbers.

Route: `/ingestionrequests/`

Method: GET

Example

```
curl \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \
   localhost:8080/ingestionrequests/
```

Ingestion Request Status

This endpoint displays the status of the specified ingestion request.

Route: `/ingestionrequests/INGESTION_REQUEST_ID_HERE`

Method: GET

Example

```
curl \
  -H'Content-Type:application/json' \
  -H'Accept:application/json' \
   localhost:8080/ingestionrequests/1
```

Classification - Easy Way

When at least one document is assigned to each leaf folder in a cabinet, it is possible to train classifiers and use them for classification. The *train* endpoint on the cabinet containing the folders is used in this case to perform the following:

1. Train a preliminary classifier for each folder in the cabinet
2. Evaluate the classifier performance, compared to earlier classifiers
3. Train an active classifier using the current best-performing classifier
4. Classify the documents within the folder using the active classifier

Train Cabinet

Route: `/cabinets/CABINET_ID_HERE/train`

Method: POST

Classification - Configurable Way

DeepNLP enables users to build hierarchical classifiers. This means that a document is assigned to a folder if, and only if it is assigned by a classifier corresponding to that folder's parent, to the classifier corresponding to the folder's parent's parent, and so on, up to the classifier for a root folder for the cabinet.
Note: A *root* folder for a cabinet has no parent.

Documents are classified by the system into folders using active classifiers. Before an active classifier can be trained for a folder, one or more preliminary classifiers must be trained as candidates to evaluate their performance. The system can select the best-performing classifier automatically, or the user can manually select the best active classifier for use.

Note that preliminary classifiers can be trained in any order. Because they are trained on documents that are manually added to a folder, they do not require the output of other classifiers for training. Their performance and accuracy can be evaluated to choose an appropriate configuration for active classifiers for a folder.

The work flow to manually configure classifiers includes:

1. One or more preliminary classifier requests are made to train a classifier for a particular folder.
2. A preliminary classifier is created by the scheduler against the existing training data for a folder. The status of the classifier can be checked through the `/prelimclassifiers/` endpoint.
3. After the classifier is created, its metrics become available through the `/metrics` endpoint.
4. A classifier request for an active classifier is made by the user that references the preliminary classifier above. Alternatively, use the `/activate` to activate an entire cabinet. The activated cabinet can:
   (a) automatically select the best-performing classifier for all folders within the cabinet
   (b) train an active classifier based on the best-performing classifier
   (c) use the trained classifier as the current active classifier for that folder

5. An active classifier is then created by the system. Its status can be checked through the `/activeclassifiers/` endpoint.
6. Because the system is capable of handling more than one active classifier for a folder, one classifier must be designated as the current active classifier. This puts them into service classifying documents in production. If the active classifier is manually created instead of created through the `/activate` endpoint, any active classifier can be selected as the current active classifier. There is no need to wait for the scheduler when selecting the current active classifier: the results are immediately.

   When these step complete, documents are classified according to the trained active classifiers.

Classifier Request

The classifier request endpoint is used to request both *preliminary* and *active* classifiers.

Route: `/folders/FOLDER_ID_HERE/requests`

Method: POST

Body Parameters:

- *wordembedding_id*: a word embedding to vectorize the documents
- *params*: a JSON string representing parameters for the model to be used. An empty string uses the defaults. See the section Classification Parameters for additional information.
- *type*: "preliminary" or "active"
- *train_fraction*: a decimal floating point number between zero and one. For each folder, if n is the number of documents in the folder, at least `train_fraction * n` documents will be used as training data for that folder
- *preliminary classifier* (optional): This parameter must be provided if and only if this request is for an active classifier, because each active classifier is created using an existing preliminary classifier

Examples

```
curl \
  -XPOST \
  -H'Content-type: application/json' \
  -H'Accept: application/json' \
   localhost:8000/folders/288/requests -d '{
             "wordembedding_id": "1",
              "params": {
             "modelType": "neuralNetwork",
             "numIterations": 6000,
             "neuralNetworkOptions" : {
               "hiddenLayers" : [20]
             }
            },
             "type": "preliminary"
  }'
```

```
curl \
 -XPOST \
 -H'Content-type: application/json'\
 -H'Accept: application/json' \
 localhost:8000/folders/288/requests -d '{
            "wordembedding_id": "1",
            "params": {
             "modelType": "neuralNetwork",
             "numIterations": 6000,
             "neuralNetworkOptions" : {
               "hiddenLayers" : [20]
             }
            },
            "type": "active",
    "preliminary_classifier": "12"
}'
```

Preliminary Classifier Retrieval

This endpoint retrieves the preliminary classifiers that have been requested and their status. Each status can be one of *requested*, *inprogress*, *failed*, or *ready*.

Route: `/prelimclassifiers/`

Method: GET

Active Classifier Retrieval

This endpoint retrieves the active classifiers that have been requested and their status. Each status can be one of *requested*, *inprogress*, *failed*, or *ready*.

Route: `/activeclassifiers/`

Method: GET

Preliminary Classifier Status

This endpoint retrieves a single requested preliminary classifier and its status. The status can be one of *requested*, *inprogress*, *failed*, or *ready*.

Route: `/prelimclassifiers/CLASSIFIER_ID_HERE`

Method: GET

Active Classifier Status

This endpoint retrieves a single requested active classifier and its status. The status can be one of *requested*, *inprogress*, *failed*, or *ready*.

Route: `/activeclassifiers/CLASSIFIER_ID_HERE`

Method: GET

Setting Current Active Classifier

This endpoint places an active classifier into service classifying real documents. Note that the `CLASSIFIER_ID` must refer to an active classifier and can be retrieved using the `/activeclassifiers/` endpoint above.

Route: `/folders/FOLDER_ID_HERE/activeclassifier/current/CLASSIFIER_ID_HERE`

Method: PUT

Embeddings

Every document is transformed to a document vector through the use of word embeddings. Every embedding used by classifiers must be registered with the system and have an *embedding ID*. Vectorized documents can be classified.

Request Word Embedding

Every word embedding, including pretrained embeddings, must be registered with the system. For pretrained embeddings, the `pretrained_url` value must be present in the request. This value instructs the system to use the existing embedding instead of training a new one.

Route: `/wordembeddingsrequest/`

Method: POST

Body Parameters:

- *collection_id*: ID of the document collection used to train this embedding, if training is required
- *type*: may be one of the following:
  - *spark_word2vec*, for a Word2Vec model trained using the specified document collection
  - *glove*, for a Glove word embedding. See the Glove algorithm website - https://nlp.stanford.edu/projects/glove/ for additional information
  - *hashingTF*, for a hashing term frequency counter
  - *pattern*, an embedding that produces a vector whose *ith* coordinate is 1 if and only if the *ith* specified regular expression matches a document, where all regular expressions are configured with the *params* field (see below)
  - *composite*, an embedding that truncates one or more of the embeddings above
- *pretrained_url*: (optional) This must be specified if and only if *type* is not *spark_word2vec*. In the case of a GloVe embedding, the URL must point to a directory that contains one of the GloVe word embeddings that are in the format downloadable from the site above. In the case of a hashingTF embedding, it must be set to the value **hashingTF**
- *params* (optional); This must be present for pattern and composite embeddings. For additional information, see the Embedding Parameters section, below.
- *description*: free-format description of the word embedding

List Word Embeddings

This endpoint lists all word embeddings together with their *status* and *ID*. The status can be *pending*. *complete*. or *error*.

Route: `/wordembeddings/`

Method: GET

Check Word Embedding Status

This endpoint checks the status and description for a specified word embedding. The status can be *pending*. *complete*. or *error*.

Route: `/wordembeddings/WORD_EMBEDDING_ID_HERE`

Method: GET

Embedding Parameters

Embedding parameters are passed using a JSON object structure described by the grammar below.

Embedding Request Parameters

- *matchOptions* (Match Parameters object, optional): This set of parameters, specified only for match embeddings, is specified below.
- *compositeOptions* (Composite Parameters object, optional): This set of parameters, specified only for composite embeddings, is specified below.

Pattern Parameters

- *patterns* (array of *string*): a non-empty list of regular expressions to match against the document

Composite Parameters

- *embeddings* (array of *int*): a non-empty list of embedding ids

Classification Parameters

Classification parameters are passed using a JSON object structure described by the grammar below

.

Classifier Request Parameters

- *modelType* (*string*, "svm" by default): can be one of the following:
  - *svm*
  - *naiveBayes*
  - *neuralNetwork*
  - *randomForest*

- *numIterations* (*int*, 10000 by default): for models that support it, this is the number of times over which the training data should be iterated during training.
  Most SparkCognition models stop training after model weight convergence, if that occurs sooner.
- *numTokens* (*int*, 500 by default): This is the number of words (or n-grams) to use to vectorize and subsequently classify each document. Tokens are used starting at the beginning of each document because tokens at the beginning of a document tend to be the most useful for classification.
- *numGrams* (*int*, 1 by default): If this is set to 1, use words for classification. If this is set to 2 or more, use n-grams. Presently n-grams should only be used with *hashingTF* embeddings. So specify numGrams > 1 only if the word embedding ID specified in the classifier request points to an embedding of type *hashingTF*.
- *svmOptions* (SVM Parameters object, optional): This set of parameters, specified only for SVM classifiers, is described below.
- *neuralNetworkOptions* (Neural Network Parameters object, optional): This set of parameters, specified only for neural networking classifiers, is described below.

SVM Parameters

Because SVM models are intrinsically binary classifiers, they must be ensembled to create a multiclass classifier. Although there are many strategies to accomplish this, DeepNLP allows two:

- *one-vs-rest*, also called *one-vs-all*, or *ova*
- *pairwise*, also called *one-vs-one* or *ovo*

One-vs-rest

If the one-vs-rest ensembling strategy is used, given K folders, `K` classifiers are built. Each classifier will be trained to distinguish documents of one class from documents of all other classes. The classifier most confident of a positive match for a document (or least confident of a negative match) is used to assign the document to its corresponding folder. A *positive match* for a classifier means that the classifier assigns the document to its corresponding folder instead of the rest, and *confidence* refers to distance from the bounding hyperplane.

Pairwise

If the pairwise ensembling strategy is used, given K folders, `K*(K-1)/2` classifiers are built. Each classifier is used to distinguish between one folder and another. All classifiers are allowed to vote on which folder a document belongs to, and the folder with the most votes wins. In the case of a tie, the only guarantee is that the choice of folder is *deterministic*. This means although the model is not guaranteed to choose any particular folder, it is guaranteed to map each document to one unique folder.

- *multiclass* (string, default "oneVsRest"): "oneVsRest" or "pairwise". see description above.

Neural Network Parameters

If a neural network is selected, a configurable feed-forward neural network will be used to map from document vectors to categories.

- *hiddenLayers* (array of *integers*, empty by default): number of elements for each layer of the neural network between the input and output layers.

Examples

Naive Bayes using the first 2000 4-grams of each document

```
        {
   "modelType": "naiveBayes",
   "numTokens": 2000,
   "numGrams": 4
}
```

Neural network with one 20-wide hidden layer trained using 15000 iterations

```
        {
   "modelType": "neuralNetwork",
   "numIterations": 15000,
   "neuralNetworkOptions": {
      "hiddenLayers":[20]}
   }
}
```

Pairwise SVMs trained using 7000 iterations over first 800 tokens from each document

```
{
   "modelType": "svm",
   "numIterations": 7000,
   "svmOptions": {
      "multiclass": "pairwise"}
   },
   "numTokens": 800
}
```

Query Documents

Documents can be searched using the `query` or `querywithkeywords` endpoint. Both take identical parameters.

Query

To retrieve a set of documents matching a query but not necessarily their keywords, the `query` endpoint can be used. Specify a cabinet, collection, or folder name to narrow the search.

Route: `/query`

Method: GET

Body Parameters:

- *query*: a query in Lucene format used to query the text of the documents.
- *collection*: name of collection of documents to search. In the case that multiple collections have the same name, all of those collections are searched.

- *cabinet*: name of cabinet whose corresponding document collection should be searched. In the case that multiple cabinets have the same name, all of their collections are searched.
- *folder*: title of folder whose documents should be searched. In the case that multiple folders have the same title, all of the folders are searched.

Example

```
curl \
 -Haccept:application/json localhost:8080/query?query=query\ & collection='test set'
```

Query With Keywords

To retrieve a set of documents matching a query but not necessarily their keywords, use the `"querywithkeywords` endpoint. It is identical to the "query" keyword except for the endpoint name.

Route: `/querywithkeywords`

Method: GET

Body Parameters:

- *query*: a query in Lucene format used to query the text of the documents
- *collection*: name of collection of documents to search. In the case that multiple collections have the same name, all of those collections will be searched
- *cabinet*: name of cabinet whose corresponding document collection should be searched. In the case that multiple cabinets have the same name, all of their collections are searched
- *folder*: title of folder whose documents should be searched. In the case that multiple folders have the same title, all of the folders are searched

Example

```
curl \
 -Haccept:application/json localhost:8080/querywithkeywords?query=query/
 \ & cabinet='default view'
```

Table

To retrieve tabular results (all spans and entities associated with the spans) of the query being passed.

Route: `/table`

Method: POST

Parameter:

- *query*: a query in Lucene format used to query the text of the documents

Body Parameters:

- *query_type*: filter or emoji
- *row_type*: span(for tabular view) or page(page view)
- *mlt*: span id for more like this

- *cursorMark*: when passed returns paginated results (* for first page and corresponding *cursorMark* returned in the subsequent responses to get next pages)
- *filters*: filters that should be applied
  - *combine_type*: AND/NOT
  -
  - *condition_values*: range of data for filtering
  - *condition_type*: between, matches etc.
- *collection*: name of collection of documents to search. In the case that multiple collections have the same name, all of those collections are searched
- cabinet: name of cabinet whose corresponding document collection should be searched. In the case that multiple cabinets have the same name, all of their collections are searched
- *folder*: title of folder whose documents should be searched. In the case that multiple folders have the same title, all of the folders are searched

Examples

```
1. curl -XPOST -Haccept:application/json -Hcontent-type:application/json \
localhost:8080/table?query_type=filter\&query='*'\&row_type=span\&collection=4\\
&mlt=8580\&cursorMark='*' -d '{"filters":[]}'


2. curl -XPOST -Haccept:application/json -Hcontent-type:application/json\
    localhost:8080/table?query_type=filter\&query='*'\&row_type=span\&collection=4\&
mlt=8580\&cursorMark='AoEmMTI0Mzkw' -d '{"filters":[]}'


3. curl -XPOST -Haccept:application/json -Hcontent-type:application/json\
    localhost:8080/table?query_type=filter\&query='*'\&row_type=span\&collection=1\
    -d '{"filters":[{"combine_type":"AND","field":"recoverable_low",\
    "condition_values":["5000","10000"], "condition_type":"between"}]}'
```

Entities

An entity is a person, place or thing in the external world referenced within a document. DeepNLP enables extracting references to entities from documents.

Create New Entity Definition

Entity types are defined for collections. Every entity type can have many occurrences within each document. The `ENTITY_TYPE_NAME` component of the path must contain only alphanumeric values and underscores.

Route: `/collections/COLLECTION_ID_HERE/entity_type/ENTITY_TYPE_NAME_HERE/`

Method:

Parameter:

- *data_type* (string, float, date, or int)

Example

```
curl \
 -XPOST \
 -Haccept:application/json localhost:8080/collections/1/entity_type/location\
 -d '{"data_type": "string"}'
```

List Entity Definitions

Entity types are defined for collections.

Route: `/collections/COLLECTION_ID_HERE/entity_type/`

Method: GET

Example

```
curl \
 -Haccept:application/json localhost:8080/collections/1/entity_type/
```

Create New Entity Occurrence

Entities can occur multiple times within a document and many times within a collection. Each occurrence must record the offsets of that occurrence and its value.

Route: `/collections/COLLECTION_ID_HERE/entity_type/ENTITY_TYPE_ID_HERE/occurrence/`

Method: POST

Parameters:

- *value* (optional): value for the entity at this location in the document. Note that the exact value does not necessarily occur within the text. For example, the letters *NY* can represent the entity "New York"
- *blob_id*: value of the ID in Solr of the portion of the document to extract. Currently, a *blob* is a page.
- *value_start_character_offset* (optional): zero-based index of the first character in the string representing the value
- *value_end_character_offset* (optional): zero-based index of the last character in the string representing the value
- *evidence_start_character_offset*: zero-based index of the first character in the string representing the evidence for the value. The evidence must provide enough context to enable a human user to verify or correct the value
- *evidence_end_character_offset*: zero-based index of the last character in the string representing the evidence for the value. The evidence must provide enough context to enable a human user to verify or correct the value

Example

```
curl \
 -XPOST \
 -Haccept:application/json localhost:8080/collections/1/entity_type/location/\
 occurrence/ -d '{
```

```
   "value": "New York",
   "span_id": "a9go8x9g8s9a",
   "value_start_character": 27,
   "value_end_character": 28,
   "evidence_start_character_offset": 20,
   "evidence_end_character_offset": 128
}'
```

List Entity Occurrences

Entities can occur multiple times within a document and many times within a collection. Each occurrence must record both the offsets of that occurrence and its value.

Route: `/collection/COLLECTION_ID_HERE/entity_type/ENTITY_TYPE_ID_HERE/occurrence/`

Method: GET

Example

```
curl \
  -Haccept:application/json localhost:8080/collections/1/entity_type/\
  location/occurrence/
```

Modify an Entity Occurrence

Entities can occur multiple times within a document and many times within a collection. Each occurrence.

Route: `/collections/COLLECTION_ID_HERE/entity_type/ENTITY_TYPE_ID_HERE/occurrence/ENTITY_OCCUR`

Method: PUT

Parameters:

- *value* (optional): value for the entity at this location in the document. Note that the exact value does not necessarily occur in the text. For example, the letters *NY* can represent the entity "New York."
- *value_start_character_offset* (optional): zero-based index of the first character in the string representing the value
- *value_end_character_offset* (optional): zero-based index of the last character in the string representing the value
- *evidence_start_character_offset* (optional): zero-based index of the first character in the string representing the evidence for the value. The evidence must provide enough context to allow a human user to verify or correct the value
- *evidence_end_character_offset* (optional): zero-based index of the last character in the string representing the evidence for the value. The evidence must provide enough context to allow a human user to verify or correct the value

Example

```
curl \
 -XPUT \
 -Haccept:application/json localhost:8080/collections\
 /1/entity_type/2/occurrence/as6s89g -d '{
   "value": "New York",
   "value_start_character_offset": 27,
   "value_end_character_offset": 28,
   "evidence_start_character_offset": 5,
   "evidence_end_character_offset": 50
}'
```

Delete an Entity Occurrence

Route: `/collections/COLLECTION_ID_HERE/entity_type/ENTITY_TYPE_ID_HERE/occurrence/ENTITY_OCCUR`

Method: DELETE

Example

```
curl \
 -XDELETE \
 -Haccept:application/json localhost:8080/collections\
 /1/entity_type/2/occurrence/as8dga9
```

Analytics on Entity

Route: `/aggregate/`

Method: GET

Body Parameters:

- *collection_id* : id of the collection to run analytics on
- *entity_type*: name of the entity to run analytics on
- *selected_input* (optional): list of span ids to filter on if nothing is passed, analytics run on entire data
- *range* (optional): this is used in a subsequent call for the actual API call to filter out rows with the range specified
- date_gap (optional): for date fields pass `month` to facet the date fields by *month* and `year` to facet by *year*. Default is 'year' if nothing passed
- min_count (optional) : facets to return if atleast min_count elements appear in each facet group. Default is '1'

Example Request

```
curl \
 -XGET \
 -s localhost:8080/aggregate/ -d '{"collection_id":1,"entity_type":"test_date"\
```

```
 ,"selected_input": [20005,20002,28423],"range":["2016/01","2018/08"],\
 "date_gap":"month"}'
```

Example Response

```
 {
 "test_date": {
  "found_count": 4,
  "null_count": 0,
  "column_type": "date",
  "found_percentage": 100,
  "null_percentage": 0,
  "histogram": {
   "2016-08": 1,
   "2017-12": 1,
   "2018-04": 1
  },
  "stats": {
   "oldest": "2016-08-30",
   "newest": "2018-04-01",
   "num_days": 579,
   "num_months": 48,
   "num_years": 1
  }
 }
}
```

Transformations

There are two main types of functions:

- row-by-row
- dataframe-at-once

```
------------------------------------------------------------------------------


# example functions:

from typing import NamedTuple



from pyspark.sql.types import StringType, Row
from slogger import getLogger
import logging
import pyspark.sql.functions as sqlfunc
from nlweb.util.transformations import AnnotatedText, annotated_text_df_type
```

```
logger = getLogger(__name__, __file__)


from nlweb.util.transformations import transform, AnnotatedText, \
    DataFrameEnvelope


class ModelTrainInput(NamedTuple):
    text : AnnotatedText


class ModelTrainOutput(NamedTuple):
    crf_ner_model : str


class SearchAndReplaceOutput(NamedTuple):
    result : str


@transform
def search_and_replace(
    text : str
) -> SearchAndReplaceOutput:
    return None


@transform(
    aggregate_in=['baz'],
    aggregate_out=['crf_ner_model'],
)
def model_train(
    df : DataFrameEnvelope[ModelTrainInput],
    baz : str
) -> ModelTrainOutput:


    print("Displaying dataframe in model_train. baz: %s" % (baz,))


    df.df.show()
    return ModelTrainOutput(
        crf_ner_model='test aggregate out'
    )
```

```
class Result(NamedTuple):
    x: str


class AnnotatedTextHolder(NamedTuple):
    x : AnnotatedText


class AnnotatedResult(NamedTuple):
    df: DataFrameEnvelope[AnnotatedTextHolder]


@transform(
    aggregate_in=['baz']
)
def ident(x: str, baz : str) -> Result:
    print("baz: %s" % (baz,))
    return Result(x=(x or 'was null'))


@transform
def ident_dataframe(
    df : DataFrameEnvelope[AnnotatedResult]
) -> DataFrameEnvelope[AnnotatedResult]:
    print("Displaying dataframe in ident_dataframe:")
    df.df.show()


    return DataFrameEnvelope(
        df=df.df.rdd.map(
            lambda r: Row(
                **{
                    k:v for k,v in r.asDict().items() if k != 'x'
                },
                x=(
                    (hasattr(r['x'], 'text') and r['x'].text)
                    or
                    AnnotatedText(
                        text='was null',
                        start=0,
                        end=0
                    )
                )
            )
        ).toDF()
```

```
    )
```

Definitions

The following are base types in Python (Solr):

str (string)

int (integer)

float (float)

datetime (date)

any (string)

enum (string) - (non-tokenized in Solr)[a]<

/ul>

Note: Although the *any* type can match any of the other types, it must be serializable to a string using the `str()` Python method to be correctly saved to Solr.

A *DeepNLP simple* type is a base type or one of the following:

- Collection of base types - A collection of a base type in Solr is a multi-valued field or an entity that is not guaranteed to occur once per span. A collection of a base type in Python is a list of that base type: that is, List[T] for type T
- Base type with an offset - Specifying that a function takes as input a base type with an offset only means that the function will be provided the offsets of an entity. If the function produces as output a base type with offsets, then the function must provide the offsets. A base type with an offset maps to the same type in Solr. The following is an example of the *type* in Python:

```
OccurenceWithOffset(Generic[T], NamedTuple):
          value: T
          begin: int
          end: int
```

  Note: If a function does not provide offsets in its output, offsets are produced automatically. This is performed through trying to find the occurrence of a value in the text of a span. If the exact occurrence is not found, the offset is set to -1.

Rules for Transformation Functions:

- The names of transformation functions must be unique.
- A row-by-row function must take as input a list of arguments of base types. It must produce one of the following:
  - a single output
  - a tuple of non-dataframe outputs.
- A dataframe-at-once function must take the one of following as input:
  - a DeepNLP dataframe envelope of an arbitrary base type

  - a DeepNLP dataframe envelope of a tuple of arbitrary base types, followed by arguments the number of which is equal to the length of the aggregate_in parameter in its 'transform' decorator
- 1. A dataframe-at-once function must produce one of the following as output:
  - a DeepNLP dataframe envelope of an arbitrary base type
  - a DeepNLP dataframe envelope of a tuple of arbitrary base types
  - a tuple containing either of the above and additional coordinates, the number of which is equal to the length of the aggregate_out parameter in its *transform* decorator.

Transform

This endpoint takes the same parameters as the *table* endpoint except for the *row_type* parameter. This parameter is always assumed to be *span*. In addition to the parameters specified below, the endpoint accepts a list of inputs and outputs. Each input and output can be a collection column, or in the case of a *dataframe-at-once* function, a named aggregate. It calls the specified function according to the following semantics:

1. A row-by-row function is called once per span. Each specified column of the span is passed to the function as an input. If the types of the inputs do not match the specified function, for example if there are more than one occurrences of a column for which the function accepts only a (scalar) base type, the span is skipped. The outputs of the function are then used to clobber all existing occurrences of the specified output columns for that row.
2. A dataframe-at-once function is called once using an entire Pyspark dataframe and any specified aggregates. Every row of the dataframe consists of the specified columns. If the types of the columns of the rows do not match the specified function, for example if there are more than one occurrences of a column for which the function accepts only a (scalar) base type, the row is filtered from the dataframe provided to the function. The dataframe is passed along with the other specified named aggregates as input. The output dataframe is used to clobber the specified output columns, and the aggregates returned by the function are then recorded.

Route: `/collections/COLLECTION_ID_HERE/transform/NAME_OF_METHOD_HERE`

Method: POST

Parameters:

- *query*: a query in Lucene format used to query the text of the documents

Body Parameters:

- input_columns: dictionary of columns to provide as input
- output_columns: dictionary of columns to clobber with the output of this method
- input_aggregations: list of aggregates to provide as input. These must be retrieved using the GET /transform endpoint
- output_aggregations (optional). Each of these names will be used as part of the final aggregation name later used to reference individual aggregations. They are of the form "aggregate_type-aggregate_prefix-date_here-random_slug", for example, "crf_ner_model-location-20180716152347-0z9ba9w". If these aren't provided, they will default to "saved."
- selected_rows: list of span ids to use as input for the transform function
- query_type: filter or emoji
- row_type : span(for tabular view) or page(page view)

- mlt : span id for more like this
- cursorMark : when passed returns paginated results (* for first page and corresponding cursorMark returned in the subsequent responses to get next pages)
- filters : filters that should be applied
  - *combine_type*: AND/NOT
  - *field*: name of the field that filter should apply to
  - *condition_values*: range of data for filtering
  - *condition_type*: between,matches etc
- *collection*: name of collection of documents to search. In the case that multiple collections have the same name, all of those collections will be searched
- *cabinet*: name of cabinet whose corresponding document collection should be searched. In the case that multiple cabinets have the same name, all of their collections will be searched
- *folder*: title of folder whose documents should be searched. In the case of multiple folders with the same title, all folders are searched

Examples

```
curl -Hcontent-type:application/json -XPOST localhost:8080/collections/1/\
    transform/model_train?  query=new+york -d '{"input_columns" :\
    {"text": "test_annotation"},\
    "output_columns" : {}, "input_aggregations": ["baz:1"], "selected_rows": []}'


  curl -Hcontent-type:application/json -XPOST localhost:8080/collections/1/\
  transform/ident?query=new+york -d '{"input_columns" : {"x": "Ans1"},\
  "output_columns" : {"x": "test_string"}, "input_aggregations":\
  ["baz:\"6\""], "output_aggregations" : [], "selected_rows": []}'


  curl -Hcontent-type:application/json -XPOST localhost:8080/collections\
  /1/transform/ident_dataframe?query=new+york -d '{"input_columns" : \
  {"x": "test_annotation"}, "output_columns" : {"x": "test_annotation"}, \
  "selected_rows": []}'
```

Retrieve Available Transformations

Route: `/collections/COLLECTION_ID_HERE/transform`

Method: GET

Parameters:

- *filter*: dictionary object containing the following fields:
  - *input_columns*: list of columns accepted as input. The "any" type does not perform any filtering: it matches a function that accepts any type for that column
  - *output_columns*: list of columns the function should provide as output. The "any" type does not perform any filtering, instead it matches a function that provides *any* type for that column
  - *input_aggregation_types*: list of aggregation prefixes. These should match the strings provided

in the aggregate_in parameter to the `@transform` decorator for the function
- *output_aggregation_types*: list of aggregation prefixes. These should match the strings provided in the aggregate_out parameter to the `@transform` decorator for the function
- *transformation_script*: glob to match Python transformation function names

Entity Extraction

When enough example entities are extracted, the system can automatically extract more. This extraction uses model training and then entity extraction, as two separate steps.

Request Training

After many occurrences of an entity type are extracted, the creation of a new model can be requested.

Route: `/collections/COLLECTION_ID_HERE/entity_type/2/model/train`

Method: POST

Example

```
curl \
 -XPOST \
 -Haccept:application/json localhost:8080/collections/1/entity_type/2/model/train
```

Request Extraction

After a model is trained, it can be used to train additional entities similar to the ones that have already been labeled. **Note**: This creates additional entities in Solr.

Route: `/collection/COLLECTION_ID_HERE/entity_type/2/model/extract`

Method: POST

Example

```
curl \
 -XPOST \
 -Haccept:application/json localhost:8080/collections/1/entity_type/2/model/extract
```

Entity Extraction Status

The status of an entity extraction model can be requested through this endpoint. There are three versions listed for a model: the edit version, the train version, and the extract version. Each version represents a globally unique version number that is incremented whenever any change is made to any occurrence of any entity.

- The edit version is the last global entity version during which any occurrence of this entity type has been changed.
- The train version is the last global entity version for which a model was trained for this entity type.
- The extract version is the last global entity version for which a model was used to extract additional entities.

The endpoint is described below:

Route: `/collections/COLLECTION_ID_HERE/entity_type/2/model/status`

Method: GET

```
curl \
 -XGET \
 -Haccept:application/json localhost:8080/collections/1/entity_type/\
 2/model/status
```

Pdf Endpoint

The process for the pdf endpoint is:

1. The UI sends a `GET` request to the *nlweb* endpoint
2. The *nlweb* sends a `POST` request to `nlpdf`
3. The *nlpdf endpoint response* is returned by the `/getpdf/` route, below

Note: Unlike all other endpoints, do not include the following headers in the request:

```
Content-type: application/json
    Accept: application/json
```

getpdf endpoint

Route: `/getpdf?docId=DOC_ID&collection=COLLECTION_ID&pageno=PAGE_NO&full_or_preview=FULL_OR_PR`

Body Parameters:

- *pageno* [Optional]: This parameter should be a number and it should be passed only if a single page is loaded.
- *full_or_preview*: This parameter can take to values, “full” or “preview”
- *highlight*: This is a json of list of dictionaries. Each dictionary should be of below format:

```
{
     “highlight”:TEXT_TO_BE_HIGHLIGHTED,
    ”color”:COLOR_CODE,
   ”type”:TYPE_OF_HIGHLIGHT
}
```

In above dictionary TYPE_OF_HIGHLIGHT can be “keywords” , “answer” , “entities” etc. In case of entities TEXT_TO_BE_HIGHLIGHTED should be an empty list ([]) because entities details are loaded from Solr.

As of now color can be either 1 , 2 , 3 , 4. Below are their RGB values and color.

```
color code : 1 , RGB :   254, 222, 135  ,  color : highlight (yellow)
color code : 2 , RGB :   177, 235, 217  ,  color :  highlight (green)
color code : 3 , RGB :   161, 201, 241  ,  color :  highlight (blue)
```

Method: GET

Example

```
curl \
 -XGET \
   localhost:8080/getpdf/collection=2 & docId=MZUWYZJ2F5WW45BPMRQXIYJP\
   OJXW65BPOBSGML3HOMXWIZLFOBXGY4BNONQWYZLTFVSGK3LPF4ZC63TPOJ3WKZ3JMF\
   XC24DFORZG63DFOVWS2ZDJOJSWG5DPOJQXIZJNMRXWG4ZPIV4HA3DPOJQXI2LPNYWWI\
   4TJNRWGS3THFVZGK43VNR2HGL2FPBYGY33SMF2GS33OFVSHE2LMNRUW4ZZNOJSXG5LMO\
   RZTEOBVFZYGIZQ= & pageno=1 & full_or_preview=full & highlight=[{"highlight":\
   "oil","color":3,"type":"keywords"},{"highlight":[],"color":\
   2,"type":"entities"}]
```

Authentication

Security Caveat

Be aware that because of the current authentication paradigm, this authentication should not be used in any implementation where security is paramount.

Create User

Route: `/add_user/`

Method: POST

Parameters:

- *user_id*: specify a user id
- *password*: specify user password

Example

```
curl -H'Content-type: application/json' -H'Accept: application/json'\
-XPOST localhost:8080/add_user -d '{"user_id":"user1","password":"test"}'
```

Login

Route: `/login/`

Method: POST

Parameters:

- *user_id*: user id value
- *password*: user password

Response: *True* or *False*

Example

```
curl -H'Content-type: application/json' -H'Accept: application/json'\
-XPOST localhost:8080/login/ -d '{"user_id":"user1" ,"password":"test"}'
```

Create Role

Route: `/add_role`

Method: POST

Parameter:

- *role*: specifies user role - either *admin* or *viewer*

Example

```
curl -H'Content-type: application/json' -H'Accept: application/json'\
-XPOST localhost:8080/add_role -d '{"role":"admin"}'
```

Add Role to User

Route: `/add_user_role`

Method: POST

Parameters:

- *user_id*: user to receive role specification
- *role*: either *admin* or *viewer*

Example

```
curl -H'Content-type: application/json' -H'Accept: application/json'\
-XPOST localhost:8080/add_user_role -d '{"user_id":"user1" , "role":"admin"}'
```

Get Roles of User

Route: `/get_roles`

Method: POST

Parameters:

- *user_id*: user id to query
- *role*: returns current role(s), *admin* or *viewer*, or `Null` if no role is set

Example

```
curl -H'Content-type: application/json' -H'Accept: application/json' \
\localhost:8080/get_roles -d '{"user_id":"user1" }'
```

Contact Support

The following resources enable you to research issues, create a support ticket, or contact SparkCognition:

- **FAQ** - Frequently Asked Questions
- Use the DeepNLP support portal - Create a support ticket and log your issue
- **Email Contact** - Send email to deepnlp_support@sparkcognition.com
- **Call Support** - The DeepNLP support line is +1-512-400-2001