Recommendation Engine Documentation

Author:
Craig Ignatowski

Contents

Module: core.py

core.tokenizeRDD(*inputRDD, sc, sw_set=None*)

Tokenizes an RDD of a list of ('_id',string) pairs

Arguments:

inputRDD : *RDD*

RDD of a list of ('_id',string) pairs to be tokenized

sc : *Spark Context*

Spark Context Environment

sw_set : *set (optional)*

A set of stopwords to be ignored when tokenizing

Returns:

An RDD of a list of tokenized ('_id',[tokens]) pairs

core.idfsRDD(*corpusRDD, sc*)

Computes the inverse-document-frequency of a set of tokenized pairs

Arguments:

corpusRDD : *RDD*

RDD of tokenized pairs to be run through inverse-document-frequency

sc : *Spark Context*

Spark Context Environment

Returns:

idfsInputWeightsBroadcast : *RDD.broadcast*

broadcast of an RDD of inverse-document-frequency weights

core.**tfidfRDD**(*tokenizedRDD, idfsInputWeightsBroadcast, sc*)

Computes the term-frequency inverse-document-frequency of a tokenized RDD

Arguments:

tokenizedRDD : *RDD*

RDD of tokenized pairs to be run through tf-idf (from **core.tokenizeRDD()**)

idfsInputWeightsBroadcast : *RDD*

broadcasted idfs RDD (from **core.idfsRDD(**)

sc : *Spark Context*

Spark Context Environment

Returns:

tfidfWeightsRDD : *RDD*

RDD of term-frequency inverse-document-frequency weights

tfidfWeightsBroadcast : *RDD.broadcast*

broadcast of **tfidfWeightsRDD**

core.**normalizeRDD**(*weightsRDD, sc*)

Normalize the weights of the terms computed from TF-IDF

Arguments:

weightsRDD : *RDD*

RDD of computed TF-IDF weights (from **core.tfidfRDD()**)

sc : *Spark Context*

Spark Context Environment

Returns:

normsBroadcast : *RDD.broadcast*

broadcast of an RDD of the normalized TF-IDF weights

core.**invertRDD**(*weightsRDD, sc*)

Inverts the ('_id',weights) pairs to (weights,'_id') pairs

Arguments:

weightsRDD : *RDD*

RDD of computed TF-IDF weights (from **core.tfidfRDD()**)

sc : *Spark Context*

Spark Context Environment

Returns:

a cached RDD of inverted pairs of the form: (weights,'_id')

core.**commonTokensRDD**(*invertedPair1RDD, invertedPair2RDD, sc*)

Collects a list of common tokens between two inverted pair RDDs

Arguments:

invertedPair1RDD : *RDD*

the first RDD of inverted pairs (from **core.invertRDD()**)

invertedPair2RDD : *RDD*

the second RDD of inverted pairs (from **core.invertRDD()**)

sc : *Spark Context*

Spark Context Environment

Returns:

a cached list of shared tokens

core.**cosineSimilarityRDD**(*commonTokens, weightsBroadcast1, weightsBroadcast2, normsBroadcast1, normsBroadcast2, sc*)

Computes the cosine similarity between two TF-IDF computed RDDs

Arguments:

commonTokens : *RDD*

a list of common tokens found between two tokenized documents

weightsBroadcast1 : *RDD.broadcast*

first broadcast of computed TF-IDF weights (from **core.tfidfRDD()**)

weightsBroadcast2 : *RDD.broadcast*

second broadcast of computed TF-IDF weights (from **core.tfidfRDD()**)

normsBroadcast1 : *RDD.broadcast*

first broadcast of an RDD of normalized TF-IDF weights (from **core.normalizeRDD()**)

normsBroadcast2 : *RDD.broadcast*

second broadcast of an RDD of normalized TF-IDF weights (from **core.normalizeRDD()**)

sc : *Spark Context*

Spark Context Environment

Returns:

a cached list of document similarities: ((doc_id1,doc_id2),cosine_similarity)

Module: jsonTools.py

jsonTools.getGroup(*json_doc*)

Determines which group a JSON document belongs to

Arguments:

json_doc : *dict*

a JSON document

Returns:

group : *string*

the name of the group the document belongs to

jsonTools.returnFields_incidents(*incidents_collection*)

Return the value of the 'summary', 'details', 'workNotes.summary', and 'workNotes.details' fields for all documents in the collection

Arguments:

incidents_collection : *mongodb.collection*

a handle to the MongoDB incidents collection

Returns:

a list containing dictionaries of the values of the fields queried

jsonTools.recFetcher(*obj*)

Recursive function which can print the values, only if they are neither a list nor a dictionary

Arguments:

obj : *object*

any object, a list of values taken from a MongoDB query in this case

Returns:

obj : *object*

an object that is neither a list nor a dictionary

jsonTools.parseMongoRecord(*mdb_record*)

Expects a record containing an '_id' key, and other important non-selection field keys such as summary, details, etc.

Arguments:

mdb_record : *dictionary*

a record returned by a MongoDB query

Returns:

parsed_record : *tuple*

an id-string tuple of the MongoDB query record

jsonTools.getText(*mdb_cursor*)

Creates a list of ('record_id','record_str') tuples from a given MongoDB query

Arguments:

mdb_cursor : *object*

a MongoDB query containing an '_id' key, and other important non-selection field keys such as summary, details, etc.

Returns:

corpus : *list*

a corpus containing a list of ('_id','str') tuples

Module: textAnalyzer.py

textAnalyzer.removeQuotes(*string*)

Remove quotation marks from an input string

Arguments:

string : *string*

input string that might have the quote "" characters

Returns:

str : *string*

a string without the quote characters

textAnalyzer.parseDatafileLine(*datafileLine, datafile_pattern*)

Parse a line of the data file using the specified regular expression pattern

Arguments:

datafileLine : *string*

input string that is a line from the data file

datafile_pattern : *string*

a regular expression describing the format of **datafileLine**

Returns:

a string parsed using the given regular expression and without quote characters

textAnalyzer.simpleTokenize(*string, split_regex=r'\W+'*)

A simple implementation of input string tokenization

Arguments:

string : *string*

input string to be tokenized

split_regex : *string (optional)*

a regular expression to be used to split the string up into tokens

Returns:

tokens : *list*

a list of tokens

textAnalyzer.**tokenize**(*string, stopwords_set*)

An implementation of input string tokenization that excludes stopwords

Arguments:

string : *string*

input string to be tokenized

stopwords_set : *set*

a set of stopwords to be ignored in the tokenization process

Returns:

tokens : *list*

a list of tokens without stopwords

textAnalyzer.**countTokens**(*vendorRDD*)

Count and return the number of tokens

Arguments:

vendorRDD : *RDD*

Pair tuple of record ID to tokenized output

Returns:

count : *int*

count of all tokens

textAnalyzer.**termFrequency**(*tokens*)

Compute the Term Frequency of the tokens

Arguments:

tokens : *list*

input list of tokens (from **textAnalyzer.tokenize()**)

Returns:

tf_dict : *dictionary*

a dictionary of tokens to their TF values

textAnalyzer.inverseDocumentFrequency(*corpus*)

Compute Inverse-Document Frequency of the tokens in the corpus

Arguments:

- **corpus** : *RDD*

  RDD of a corpus containing a set of all tokens found in the query

Returns:

- **idfRDD** : *RDD*

  RDD of (token, IDF value) from the corpus

textAnalyzer.tfidf(*tokens, idfs*)

Compute the Term Frequency-Inverse Document Frequency of the tokens in the corpus

Arguments:

- **tokens** : *list*

  input list of tokens (from **textAnalyzer.tokenize()**)

- **idfs** : *dictionary*

  a dictionary of records to their IDF values

Returns:

- **tfidf_dict** : *dictionary*

  a dictionary of records to their TF-IDF values

textAnalyzer.invert(*record*)

Invert a list of ('_id', [tokens]) to a list of ('token', '_id')

Arguments:

- **record** : *tuple*

  a tuple of the form: ('_id', [token_vector])

Returns:

- **inverted_record** : *list*

  a list of tuples of the form: ('token', '_id')

textAnalyzer.swap(*record*)

Swap (token, ('docId', 'queryId')) to (('docId', 'queryId'), 'token')

Arguments:

- **record** : *tuple*

  a tuple of the form: (token, ('docId', 'queryI'))

Returns:

- **swapped_record** : *tuple*

  a tuple of the form:(('docId', 'queryI'), 'token')

Class: textAnalyzer.cosineSimilarity

textAnalyzer.cosineSimilarity.dotprod(*a*, *b*)

Compute the dot product between two values

Arguments:

a : *dictionary*

first dictionary of record to value

b : *dictionary*

second dictionary of record to value

Returns:

dotProd : *float*

result of the dot product of the two input dictionaries

textAnalyzer.cosineSimilarity.norm(*a*)

Compute the normalization of the input dictionary

Arguments:

a : *dictionary*

a dictionary of record to value

Returns:

norm : *dictionary*

a dictionary of tokens to their normalized TF values

textAnalyzer.cosineSimilarity.cossim(*a*, *b*)

Compute the cosine similarity between two values

Arguments:

a : *dictionary*

first dictionary of record to value

b : *dictionary*

second dictionary of record to value

Returns:

cossim : *float*

dot product of two dictionaries divided by the norm of the first dictionary and then by the norm of the second dictionary

textAnalyzer.cosineSimilarity.**cosineSimilarity**(*string1, string2, idfsDictionary*)

Compute the cosine similarity between two values

Arguments:

string1 : *string*

first string

string2 : *string*

second string

idfsDictionary : *dictionary*

a dictionary of IDF values

Returns:

cossim : *float*

cosine similarity value

textAnalyzer.cosineSimilarity.**fastCosineSimilarity**(*record, inputWeightsBroadcast, databaseWeightsBroadcast, inputNormsBroadcast, databaseNormsBroadcast*)

Compute Cosine Similarity using Broadcast variables

Arguments:

record : *tuple*

a tuple of the form: ((’docId’, ’queryId’), ’token’)

inputWeightsBroadcast : *RDD.broadcast*

broadcasted input TF-IDF weights RDD

databaseWeightsBroadcast : *RDD.broadcast*

broadcasted database TF-IDF weights RDD

inputNormsBroadcast : *RDD.broadcast*

broadcasted input normalized weights RDD

databaseNormsBroadcast : *RDD.broadcast*

broadcasted database normalized weights RDD

Returns:

cossim : *tuple*

a computed cosine similarity tuple of the form: ((’docId’, ’queryId’), ’cosineSimilarity-Value’)

Module: weighting.py

weighting.grabWeightingQueryFormat(*mdb_collectionType="incidents"*)

Determines the query format that the recommendation engine will use for weighting based on the collection/ticket type

Arguments:

mdb_collectionType : *string (optional)*

the type of ticket being searched and recommended against

Returns:

weighting_query_format : *dictionary*

format of the mongo query for a specific ticket type

weighting.applyWeights(*record, firstPass_overallWeight=1.0*)

Applies the weight of first pass of the recommendation engine to the second pass of the engine

Arguments:

record : *tuple*

tuple of the form: (('docId', 'queryId'), ('firstPass_score', 'secondPass_score'))

firstPass_overallWeight : *float (optional)*

the weighting effect that the first pass, overall, has compared to the second pass

Returns:

jointWeight : *tuple*

tuple of the form: (('docId', 'queryId'), 'jointWeight')

weighting.parseMongoRecordRelevancy(*mdb_record*)

Grabs the relevancy score from a MongoDB record

Arguments:

mdb_record : *dictionary*

a MongoDB record

Returns:

rlvcTuple : *tuple*

a tuple of the form: ('recId', 'rcRlvcScore')

weighting.getRelevancy(*mdb_cursor*)

Generate a list of document Ids with their relevancy scores

Arguments:

mdb_cursor : *object*

a MongoDB query containing an '_id' key, and other important non-selection field keys such as summary, details, etc.

Returns:

rlvcList : *list*

a list of tuples of relevancy scores of the form: ('_id', 'rlvcScore')

weighting.parseRecommendation(*recommendations*)

Grabs the query id of a document and its recommendation score

Arguments:

recommendations : *list*

a list of tuples of ids and their recommendation scores of the form: (('recId', 'queryId'), 'recScore')

Returns:

reducedRecommendations : *list*

a list of tuples of just the query ids and their recommendation scores of the form: ('queryId', 'recScore')

weighting.applyRlvc(*record, overallRlvcWeight=1.0*)

Applies the weight of the relevancy score to the joint similarity score of the recommendation engine

Arguments:

record : *tuple*

a tuple of a query Id and its similarity and relevancy scores of the form: ('queryId', ('simScore', 'rlvcScore'))

overallRlvcWeight : *float (optional)*

the weighting effect that the relevancy score, overall, has compared to the similarity score

Returns:

recommendationScore : *tuple*

a tuple of the query Id and its calculated recommendation score of the form: (queryId, recScore)

Module: parseAndInsert.py

parseAndInsert.paiManyWithCollection(*infile, mdb_collection*)

Given a file containing JSON documents, this function parses the contents of the file into separate JSON documents, inserts them into a given MongoDB collection, and returns a list of MongoDB ObjectIds associated with the inserted documents

Arguments:

infile : *string*

location of a text file containing JSON documents

mdb_collection : *mongodb.collection*

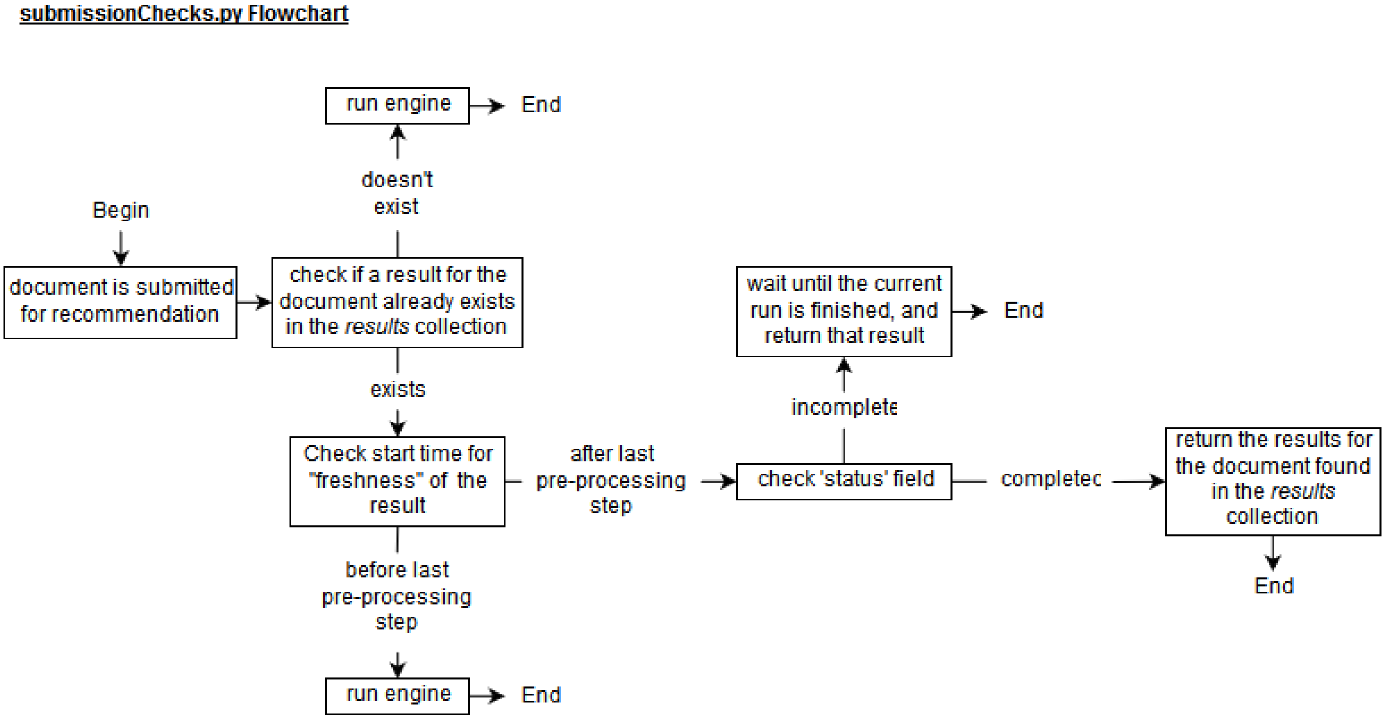
a mongoDB collection pointer

Returns:

insertedIds : *list*

a list of the MongoDB ObjectIds assigned to the documents that were submitted

Module: submissionChecks.py


submissionChecks.py Flowchart
Begin
document is submitted for recommendation
check if a result for the document already exists in the *results* collection
doesn't exist
run engine
End
exists
Check start time for "freshness" of the result
before last pre-processing step
run engine
End
after last pre-processing step
check 'status' field
incomplete
wait until the current run is finished, and return that result
End
completed
return the results for the document found in the *results* collection
End

submissionChecks.checkResultExists(*inputId*, *mdb_results_collection*)

This function checks if recommendation results for a given input MongoDB ObjectId already exists within the *recResult* collection

Arguments:

inputId : *string*

the string within the ObjectId() of the MongoDB ObjectId, i.e. ObjectId(**inputId**)

mdb_results_collection : *mongodb.collection*

a mongoDB collection pointer to the *recResult* collection

Returns:

resultExists : *boolean*

it is whether or not a result for the given **inputId** already exists in the *recResult* collection

submissionChecks.**checkResultStatus**(*inputId, mdb_results_collection*)

If a result already exists in the *recResult* collection, then this function checks the 'status' field of the *recResult* document to see if the status of the result is showed as 'completed', i.e. {'status':3}

Arguments:

inputId : *string*

the string within the ObjectId() of the MongoDB ObjectId, i.e. ObjectId(**inputId**)

mdb_results_collection : *mongodb.collection*

a mongoDB collection pointer to the *recResult* collection

Returns:

statusCompleted : *boolean*

if a result exists in the *recResult* collection, then it is whether or not the 'status' field shows the result as 'completed'

submissionChecks.**checkResultTiming**(*inputId, mdb_results_collection, lastPreProcessTime*)

If a result already exists in the *recResult* collection and it is also shown to be a 'completed' ticket, then this function checks that the latest result returned was generated *after* the last preProcess run

Arguments:

inputId : *string*

the string within the ObjectId() of the MongoDB ObjectId, i.e. ObjectId(**inputId**)

mdb_results_collection : *mongodb.collection*

a mongoDB collection pointer to the *recResult* collection

lastPreProcessTime : *float*

the timestamp, in seconds of the last preProcess run

Returns:

completedAfterLastPreProcess : *boolean*

if a result exists in the *recResult* collection, and the result's 'status' field is shown as 'completed', then it is whether or not the latest result returned was generated after the last preProcess run

Module: algorithms.py

algorithms.**muCalculator**(*lowerBound=7200., upperBound=28800.*)

Calculates the value of μ for a gaussian curve

Arguments:

lowerBound : *float (optional)*

the time, in seconds, of the lower bound of the gaussian

upperBound : *float (optional)*

the time, in seconds, of the upper bound of the gaussian

Returns:

mu : *float*

the time value of the center of the peak for the gaussian

μ is the center of the peak for the Gaussian curve given by the equation:

$$\mu = \frac{b - a}{2} + a$$

where a = lowerBound, and b = upperBound.

algorithms.sigmaCalculator(*mu,boundValue=7200., boundsWeightConfig=1.1, peakWeightConfig=1.4*)

Calculates the value of σ for a gaussian curve

Arguments:

mu : *float*

the time value of the center of the peak for the gaussian

boundValue : *float (optional)*

the time, in seconds, of either the lower or upper bound

boundWeightConfig : *float (optional)*

the weight the gaussian will return for boundValue

peakWeightConfig : *float (optional)*

the weight the gaussian will return for μ

Returns:

sig : *float*

the value of sigma that will cause the gaussian to return boundsWeightConfig at both the upper and lower bound

σ is the variance of the Gaussian curve. It describes how far a set of random number are spread out from the mean (μ), given by the equation:

$$\sigma = \frac{x - \mu}{\sqrt{-2 \cdot \ln\left(\frac{w-1}{z-1}\right)}}$$

where x = **boundValue**, w = **boundsWeightConfig**, z = **peakWeightConfig**, and μ = the μ generated by **algorithms.muCalculator()**.
Note: ln is the natural log, a logarithm with a base of the mathematical constant e: $\ln(x) = \log_e(x)$.

algorithms.gaussian(*time, mu, sigma, peakWeightConfig=1.4*)

Returns the value of the gaussian curve for a given time

Arguments:

time : *float*

the time, in seconds, that a gaussian weight will be calculated for

mu : *float*

the time value of the center of the peak for the gaussian

sigma : *float*

the value of sigma that will cause the gaussian to return boundsWeightConfig at both the upper and lower bound

peakWeightConfig : *float (optional)*

the weight the gaussian will return for μ

Returns:

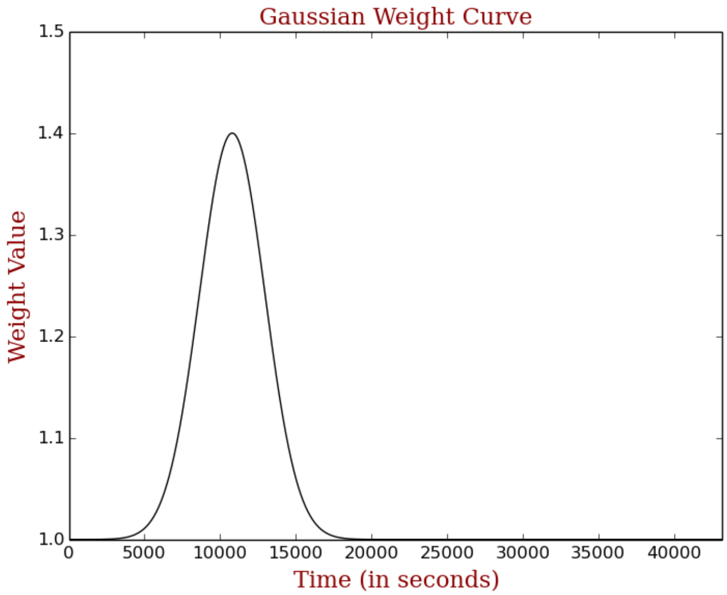
timeWeight : *float*

the weight the gaussian returns for a given input time

The Gaussian is a normal distribution of a dataset, given by the equation:

$$y = 1 + (z - 1) \cdot \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

where $y =$ **timeWeight** , $x =$ **time**, $z =$ **peakWeightConfig**, $\mu =$ the μ generated by **algorithms.muCalculator()**, and $\sigma =$ the σ generated by **algorithms.sigmaCalculator()**.
Note: exp() is the exponential function of the mathematical constant e: $\exp(x) = e^x$.



The above figure shows the curve that **algorithms.gaussian()** follows.
For this particular curve, **mu** = 10800, **sigma** ≈ 2162.02034, and **peakWeightConfig** = 1.4.

algorithms.**priorityLevelWeighting**(*sortedPriority, priorityLevelsMaxWeight=1.5, numPriorityLevels=4*)

Returns the weight a sortedPriority value has for a grouping with x levels of priority

Arguments:

sortedPriority : *int*

the sorted value of a ticket's priority level, the lower the value, the higher the priority (0 being the highest priority)

priorityLevelsMaxWeight : *float (optional)*

the value of the weight that the message will have if its sorted priority value is 0

numPriorityLevels : *int (optional)*

the number of levels of priority that the group the message belongs to has

Returns:

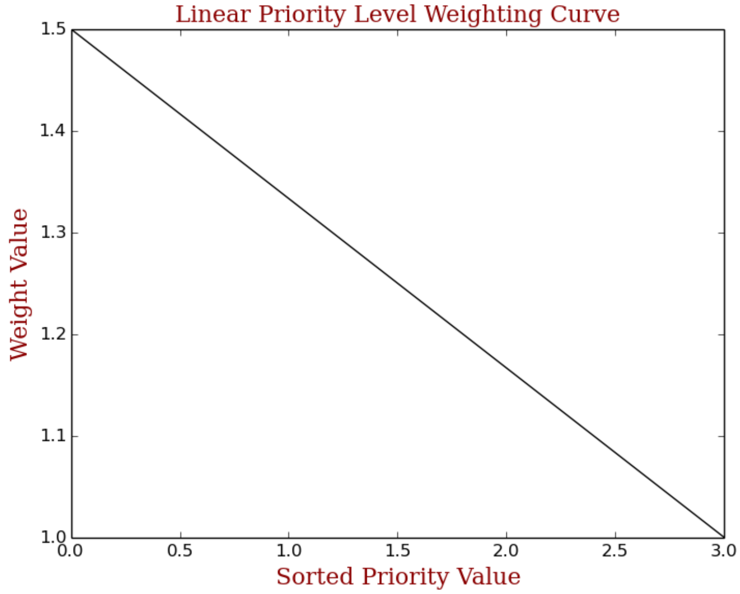
priorityWeight : *float*

the value of the weight of the message's priority level

The linear fit of the weight of the priority levels is given by the equation:

$$y = w - \left(\frac{w-1}{z-1}\right) \cdot x$$

where $x =$ **sortedPriority**, $w =$ **priorityLevelsMaxWeight**, and $z =$ **numPriorityLevels**



The above figure shows the curve that **algorithms.priorityLevelWeighting()** follows. For this particular curve, the optional variables were kept at their default values.

algorithms.**logisticsGrowth**(*numWorkNotes, upperNumWorkNotesBound=20, minNumWorkNotes=2, upperNumBoundWeight=1.4*)

Returns the value of a logistics curve for a given number of work notes to be used as the relevancy weighting due to the number of work notes in the document

Arguments:

numWorkNotes : *int*

the number of work notes found in the document

upperNumWorkNotesBound : *int (optional)*

number of work notes after which diminishing returns becomes a serious factor and the value will not grow much higher than the value of upperNumBoundWeight

minNumWorkNotes : *int (optional)*

the minimum number of work notes that need to exist in the document before work notes are taken into account for relevancy weighting

upperNumBoundWeight : *float (optional)*

the value of the weight work notes has on the relevancy score at upperNumWorkNotesBound, the soft-cap of the weight

Returns:

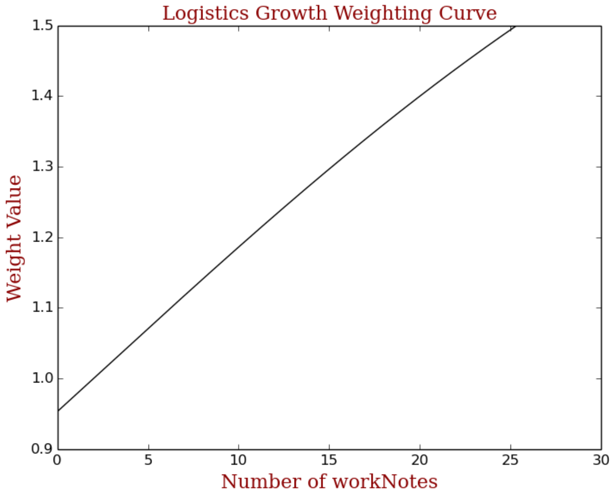
numWorkNotesWeight : *float*

the value of the weight of the number of work notes found in the document

The logistics growth curve of the weight of the number of work notes found in the document is given by the equation:

$$y = \frac{2}{1 + \exp\left[\frac{\ln\left(\frac{2-w}{w}\right)}{a-z} \cdot (x - z)\right]}$$

where y = **numWorkNotesWeight**, x = **numWorkNotes**, a = **upperNumWorkNotesBound**, z = **minNumWorkNotes**, and w = **upperNumBoundWeight**.



The above figure shows the curve that **algorithms.logisticsGrowth()** follows.
For this particular curve, the optional variables were kept at their default values.

algorithms.logisticsGrowth2(*numTokens, upperNumTokensBound=150., minNumTokens=10., upperBoundWeight=1.4*)

Returns the value of a logistics curve for a given token count to be used as the relevancy weighting due to the number tokens in a given field

Arguments:

numTokens : *int*

the number of tokens found in the field

upperNumTokensBound : *float (optional)*

number of tokens after which diminishing returns becomes a serious factor and the value will not grow much higher than the value of upperBoundWeight

minNumTokens : *float (optional)*

the minimum number of tokens that need to exist in the field before token count is taken into account for relevancy weighting

upperBoundWeight : *float (optional)*

the value of the weight token count has on the relevancy score at upperNumTokensBound, the soft-cap of the weight

Returns:

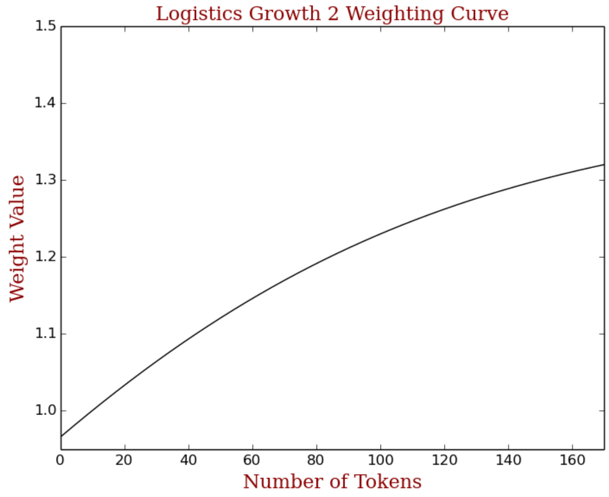
numTokensWeight : *float*

the value of the weight of the number of tokens found in the field

The second version of the logistics growth curve of the weight of the number of tokens found in a given field, after the removal of stopwords, is given by the equation:

$$y = \frac{w}{1 + (w-1) \cdot \exp\left[(z-x) \cdot \left(\frac{\ln\left(w-1\right) - \ln\left(3(w-1)^2 + 4(w-1)\right)}{z-a}\right)\right]}$$

where y = **numTokensWeight**, x = **numTokens**, a = **upperNumTokensBound**, z = **minNumTokens**, and w = **upperBoundWeight**.



The above figure shows the curve that **algorithms.logisticsGrowth2()** follows. For this particular curve, the optional variables were kept at their default values.

algorithms.**logGrowth**(*inputNum*, *minNum=2*, *upperNumBound=20*, *upperBoundWeight=1.4*)

Returns the value of a logarithmic growth curve for a given input value

Arguments:

inputNum : *int*

the input value for which a relevancy weight will be calculated

minNum : *int (optional)*

the minimum value needed as input in order for a weight to be calculated

upperNumBound : *int (optional)*

the value at which diminishing returns becomes a serious factor and the value will not grow much higher than the value of upperBoundWeight

upperBoundWeight : *float (optional)*

the value of the weight that will be returned for upperNumBound as the input, the soft-cap of the weight

Returns:

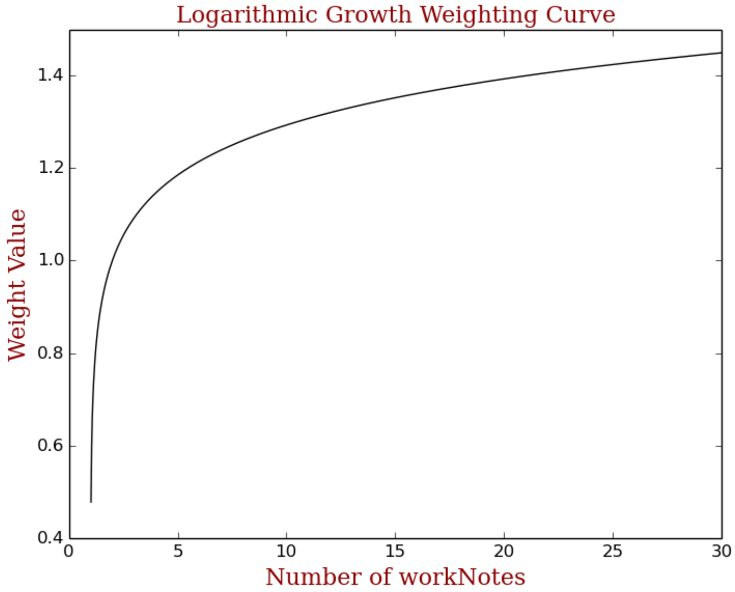
logWeight : *float*

the value of the logarithmic weight for the given input

The logarithmic growth curve of the weight of a given parameter is given by the equation:

$$y = 1 + \frac{\ln\big(x - (z-1)\big) \cdot (w-1)}{\ln(a)}$$

where $y =$ **logWeight**, $x =$ **inputNum**, $a =$ **upperNumBound**, $z =$ **minNum**, and $w =$ **upperBoundWeight**.



The above figure shows the curve that **algorithms.logGrowth()** follows.
For this particular curve, the optional variables were kept at their default values.

algorithms.exponential(*inputNum, minNum=0, upperNumBound=4, upperBoundWeight=1.5*)

Returns the value of an exponential growth curve for a given input value

Arguments:

inputNum : *int*

the input value for which a relevancy weight will be calculated

minNum : *int (optional)*

the minimum value needed as input in order for a weight to be calculated

upperNumBound : *int (optional)*

the value at which the weight is the maximum value, upperBoundWeight

upperBoundWeight : *float (optional)*

the value of the weight that will be returned for upperNumBound as the input, the hard-cap of the weight

Returns:

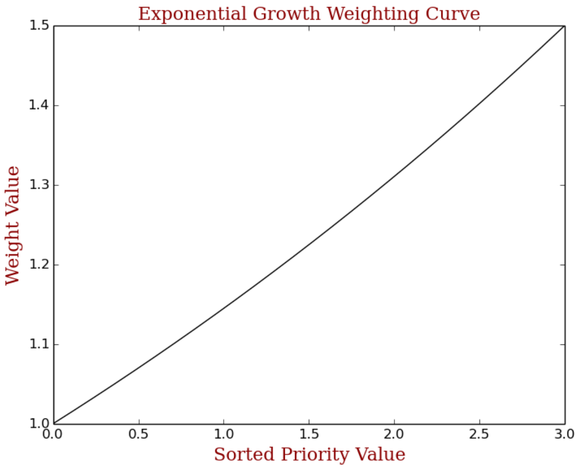
exponentialWeight : *float*

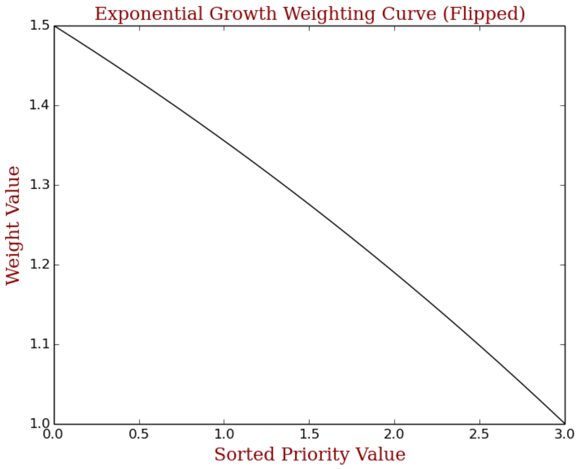
the value of the exponential weight for the given input

The exponential growth curve of the weight of a given parameter is given by the equation:

$$y = \exp\left[\frac{\ln(w)}{a - z - 1} \cdot (x - z)\right]$$

where y = **exponentialWeight**, x = **inputNum**, a = **upperNumBound**, z = **minNum**, and w = **upperBoundWeight**.





The above figures show the curve that **algorithms.exponential()** follows. The figure on the right shows the curve of the weighting after the **sortedPriority** value has been flipped, so that 0 represents a *critical* priority and a 3 represents a *low* priority, which is the proper representation of the **sortedPriority** value. For these particular curves, the optional variables were kept at their default values.

algorithms.cosineGrowth(*sortedPriority, numPriorityLevels=4, maxPriorityWeight=1.5*)

Returns the weight a sortedPriority value has for a grouping with x levels of priority, using a cosine function as the curve

Arguments:

sortedPriority : *int*

the sorted value of a ticket's priority level, the lower the value, the higher the priority (0 being the highest priority)

numPriorityLevels : *int (optional)*

the number of levels of priority that the group the message belongs to has

maxPriorityWeight : *float (optional)*

the value of the weight that the message will have if its sorted priority value is 0

Returns:

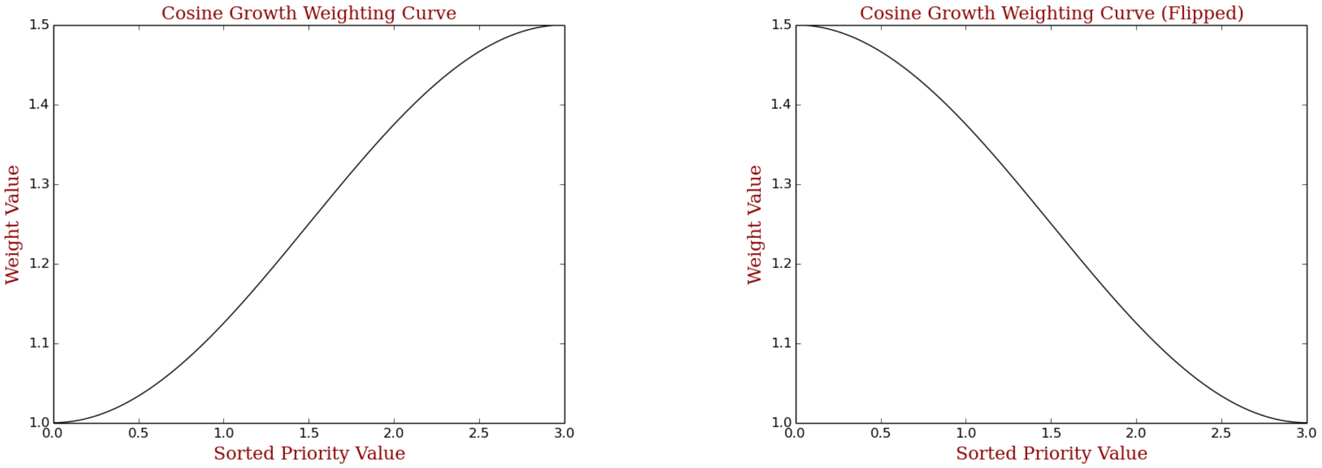
cosinePriorityWeight : *float*

the value of the cosine-driven weight of the message's priority level

The cosine growth curve of the weight of the priority level is given by the equation:

$$y = 1 + \left(\frac{w-1}{2}\right) \cdot \left(1 + \cos\left[\pi \cdot \left(\frac{(l-1)-x}{l-1}\right) - 1\right]\right)$$

where y = **cosinePriorityWeight**, x = **sortedPriority**, l = **numPriorityLevels**, and w = **maxPriorityWeight**.



The above figures show the curve that **algorithms.cosineGrowth()** follows. The figure on the right shows the curve of the weighting after the **sortedPriority** value has been flipped, so that 0 represents a *critical* priority and a 3 represents a *low* priority, which is the proper representation of the **sortedPriority** value. For these particular curves, the optional variables were kept at their default values.

algorithms.**inverseWeighting**(*weight_list*)

Calculates the total relevancy score a document will have Rather than multiplying the weights together, this will do the inverse, where applying additional weights will have diminishing returns on the score, approaching the value of 1, or 100% additional weighting. With this method, however, the total score will always be ≥ 0 and < 1

Arguments:

weight_list : *list*

a list of weights to be concatenated into the relevancy score

Returns:

weighted_score : *float*

the result of inverse concatenation of the weights

The diminishing returns approach of applying weights to the score is given by the equation:

$$C = 1 - \prod_{i=1}^{n} (2 - W_i)$$

where $C =$ **weighted_score**, $[W_1, W_2, ..., W_n]$ are the weights found in **weight_list**, and n is the length of **weight_list**.

Module: mongoBall.py

Class: mongoBall.mongoBall

mongoBall.mongoBall.__init__(*self, database='test', client='mongodb://localhost', port=27017, passProtected=False, userName='user', userPass='pass'*)

Initializes the MongoDB setup object

Arguments:

database : *string (optional)*

name of the Mongo database

client : *string (optional)*

name of the Mongo database client

port : *int (optional)*

port number to connect to the Mongo database client

passProtected : *boolean (optional)*

whether or not the Mongo database requires a password to access

userName : *string (optional)*

a username to be used to connect to a password-protected Mongo database

userPass : *string (optional)*

a password for the username to be used to connect to a password-protected Mongo database

Returns:

None

mongoBall.mongoBall.getDatabase(*self*)

Return the name of the Mongo Database

Arguments:

None

Returns:

self.database

mongoBall.mongoBall.getClient(*self*)

Return the name of the Mongo client host

Arguments:

None

Returns:

self.client

mongoBall.mongoBall.getPort(*self*)

Return the port number

Arguments:

None

Returns:

self.port

mongoBall.mongoBall.stat(*self*)

Prints the value of self.database, self.client, and self.port

Arguments:

None

Returns:

print statements of self.database, self.client, and self.port

mongoBall.mongoBall.connection(*self*)

Establish a connection to the database

Arguments:

None

Returns:

mdb_connection : *mongodb*

an established connection to a Mongo database

mongoBall.mongoBall.db(*self*)

Get a handle to the database

Arguments:

None

Returns:

mdb : *mongodb.database*

an authenticated handle on a Mongo database

mongoBall.mongoBall.collection(*self, collection*)

Access collection 'collection'

Arguments:

collection : *string*

name of the Mongo DB collection

Returns:

mdb_collection : *mongodb.collection*

an authenticated handle on a MongoDB collection

Class: mongoBall.mongoURIBall

Extends mongoBall.mongoBall

mongoBall.mongoBall.__init__(*self, mongoURI="mongodb://user:pass@localhost:27017/test"*)

Initializes the MongoDB setup object

Arguments:

mongoURI : *string (optional)*

a full MongoDB URI containing all of the credentials necessary to connect to a password-protected Mongo database

Returns:

None

mongoBall.mongoBall.connection(*self*)

Establish a connection to the database through a URI

Arguments:

None

Returns:

mdb_connection : *mongodb*

an established connection to a Mongo database via URI

mongoBall.mongoBall.db(*self*)

Get a handle to the default database in the URI

Arguments:

None

Returns:

mdb : *mongodb.database*

an authenticated handle on a Mongo database via URI

Class: mongoBall.recConfigURI

Extends mongoBall.mongoURIBall

mongoBall.mongoBall.__init__(*self, mongoURI="mongodb://user:pass@localhost:27017/test", collection='recConfig', groupingId='default'*)

Initializes the *recConfig* collection setup and grabs a handle on it

Arguments:

mongoURI : *string (optional)*

a full MongoDB URI containing all of the credentials to connect to a Mongo database

collection : *string (optional)*

name of the collection to get a handle on (default: "recConfig")

groupingId : *string (optional)*

the grouping Id for which the *recConfig* collection contains a particular document of configurables

Returns:

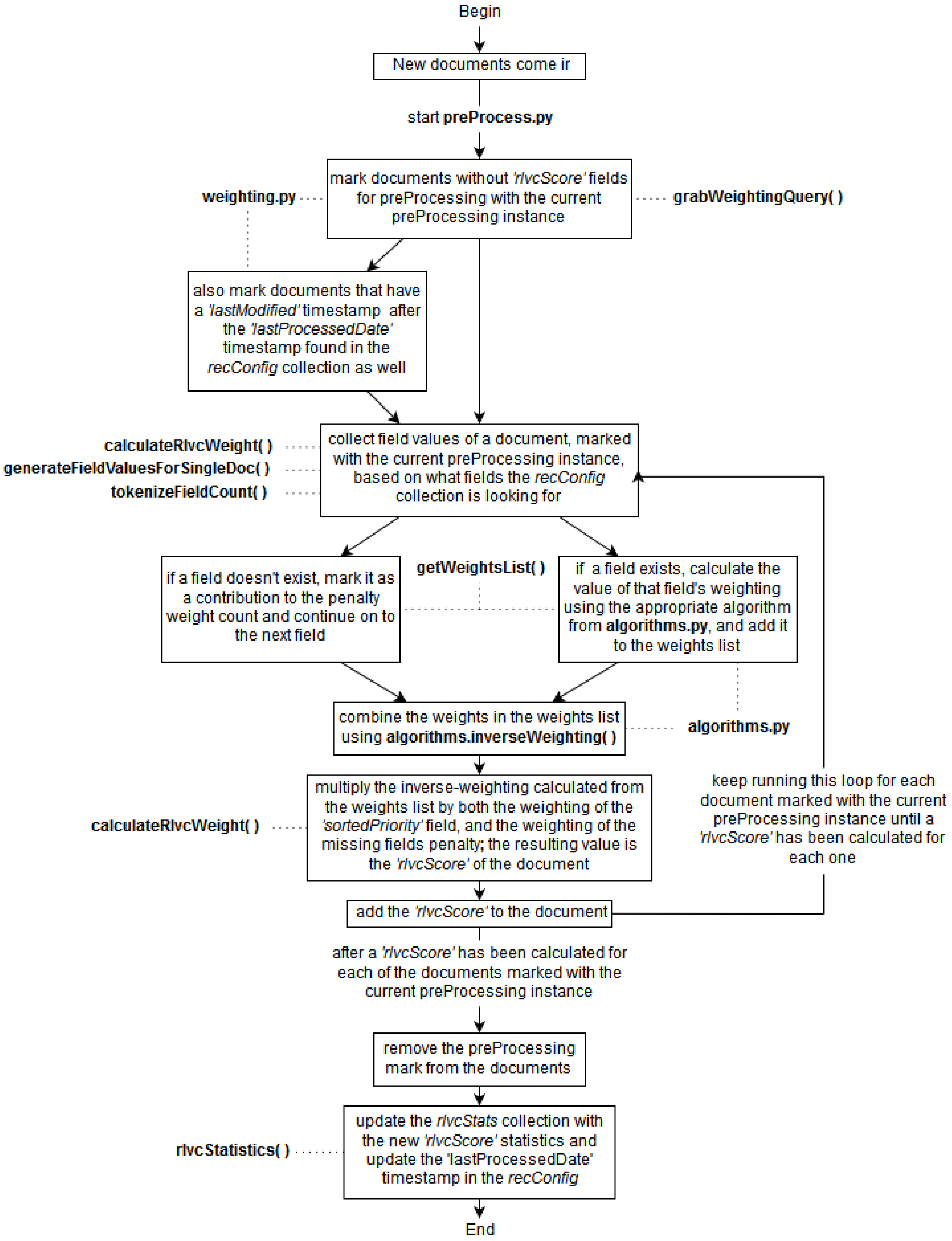
None

The *recConfig* collection contains a unique document of configurables for a given **groupingId** for use in **submission.py** and **preProcess.py**. Each document has the following layout:

```
{   "_id" : ObjectId("573b879d7f9ee95f53d66cd4"),
    "filterTopPercent" : 0.2,
    "groupingId" : "default",
    "lastProcessedDate" : datetime.datetime(1970, 1, 1, 0, 0),
    "overallRlvcWeight" : 0.6,
    "recEngineInputLimit" : 100000
    "recTTL" : 3600,
    "recTotalOutput" : 5,
    "rlvcNumPriorityLevels" : {
        "configValue" : 4,
        "weight" : 1.5},
    "rlvcNumSummaryWords" : {
        "configValue" : 1,
        "weight" : 1.3},
    "rlvcNumDetailsWords" : {
        "configValue" : 15,
        "weight" : 1.3},
    "rlvcNumWorkNotes" : {
        "configValue" : 2,
        "weight" : 1.5},
    "rlvcNumWorkNoteSummaryWords" : {
        "configValue" : 10,
        "weight" : 1.4},
    "rlvcNumWorkNoteDetailsWords" : {
        "configValue" : 10,
        "weight" : 1.4},
    "rlvcTimeRangeLowerBound" : {
        "configValue" : 7200,
        "weight" : 1.1},
    "rlvcTimeRangeUpperBound" : {
        "configValue" : 28800,
        "weight" : 1.4},}
```

Relevancy Engine: preProcess.py

preProcess.py Flowchart


Begin
New documents come ir
start preProcess.py
mark documents without 'rlvcScore' fields for preProcessing with the current preProcessing instance
weighting.py
grabWeightingQuery( )
also mark documents that have a 'lastModified' timestamp after the 'lastProcessedDate' timestamp found in the recConfig collection as well
calculateRlvcWeight( )
generateFieldValuesForSingleDoc( )
tokenizeFieldCount( )
collect field values of a document, marked with the current preProcessing instance, based on what fields the recConfig collection is looking for
if a field doesn't exist, mark it as a contribution to the penalty weight count and continue on to the next field
getWeightsList( )
if a field exists, calculate the value of that field's weighting using the appropriate algorithm from algorithms.py, and add it to the weights list
combine the weights in the weights list using algorithms.inverseWeighting( )
algorithms.py
calculateRlvcWeight( )
multiply the inverse-weighting calculated from the weights list by both the weighting of the 'sortedPriority' field, and the weighting of the missing fields penalty; the resulting value is the 'rlvcScore' of the document
keep running this loop for each document marked with the current preProcessing instance until a 'rlvcScore' has been calculated for each one
add the 'rlvcScore' to the document
after a 'rlvcScore' has been calculated for each of the documents marked with the current preProcessing instance
remove the preProcessing mark from the documents
rlvcStatistics( )
update the rlvcStats collection with the new 'rlvcScore' statistics and update the 'lastProcessedDate' timestamp in the recConfig
End

preProcess.py is the core application that is submitted to spark-submit for the relevancy engine. This application is called in the following way:

```
~$ preProcess.py groupingId mongoDB_uri <mongoDB_collection>
                 <rlvcStats_collection> <recConfig_uri>
                 <recConfig_collection>
```

Arguments:

groupingId : *string*

the grouping Id that the input document belongs to in the Mongo database

mongoDB_uri : *string*

URI of the Mongo database that contains the documents to be recommended against

mongoDB_collection : *string (optional)*

name of the MongoDB collection that contains the documents that will be recommended (default: "message")

rlvcStats_collection : *string (optional)*

name of the MongoDB collection that contains the statistics of the 'rlvcScore's of the given **groupingId** (default: "rlvcStats")

recConfig_collection : *string (optional)*

name of the MongoDB collection that contains user-configurable variables for use in both the recommendation engine and the pre-processing engine (default: "recConfig")

recConfig_uri : *string (optional)*

URI of the Mongo Database that contains the user-configurable variables documents (default: same as mongoDB_uri)

preProcess.**grabWeightingQuery**(*groupingId, mdb_collection, mdb_collectionType="incidents", withRlvcScore=False*)

Grabs the MongoDB query format for incident files and creates a handle on the query

Arguments:

groupingId : *string*

The grouping Id that the input document belongs to in the Mongo Database

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

mdb_collectionType : *string (optional)*

the type of ticket being searched and recommended against

withRlvcScore : *boolean (optional)*

whether or not the query looks for documents for which a 'rlvcScore' field exists

Returns:

weight_fields_query : *mongodb.query*

a MongoDB query across a given **groupingId**, formatted for 'incident' type files, where a 'rlvcScore' field either exists or does not, depending on the **withRlvcScore** flag

preProcess.**tokenizeFieldCount**(*field_name, doc, sw_set*)

Counts the number of tokens in a given field, excluding tokens listed in a given stopword set

Arguments:

field_name : *string*

the name of the document field being tokenized and counted

doc : *dictionary*

a single MongoDB document

sw_set : *set*

a set of stopwords to be ignored when tokenizing

Returns:

tokenCount : *int*

the number of stopword-excluded tokens found in the given field for the given document

preProcess.**generateFieldValuesForSingleDoc**(*doc, sw_set*)

Generates a dictionary of field values for the *recConfig* parameters

Arguments:

doc : *dictionary*

a single MongoDB document

sw_set : *set*

a set of stopwords to be ignored when tokenizing

Returns:

configResults : *dictionary*

a dictionary of field values and other field parameters taken from the given document to be compared with the parameters in the *recConfig* document

preProcess.**getWeightsList**(*configResults, recConfig_handle*)

returns a list of weights based off the configResults dictionary compared with the *recConfig* parameters of the given **groupingId**

Arguments:

configResults : *dictionary*

a dictionary of field values and other field parameters taken from a given document, generated by **preProcess.generateFieldValuesForSingleDoc()**

recConfig_handle : *mongoBall.recConfigURI*

a handle on a *recConfig* document in the *recConfig* collection and its parameters for the given **groupingId**

Returns:

weightsList : *list*

a list of each **configResults** field's calculated weight based on the *recConfig* parameters of the given **groupingId**

penalty : *float*

a penalty weight to be applied to the 'rlvcScore' for missing fields

prePreprocess.calculateRlvcWeight(*doc, sw_set, recConfig_handle*)

Calculates the relevancy score for a given document based on weights and penalties calculated in **preProcess.getWeightsList()**, and the 'priority' of the document

Arguments:

doc : *dictionary*

a single MongoDB document for which a relevancy score will be calculated

sw_set : *set*

a set of stopwords to be ignored when tokenizing

recConfig_handle : *mongoBall.recConfigURI*

a handle on a *recConfig* document in the *recConfig* collection and its parameters for the given **groupingId**

Returns:

rlvcScore : *float*

a score given to a document describing how relevant it is to the database based on parameters given by the *recConfig* document of the given document's **groupingId**

preProcess.rlvcStatistics(*topNPercent, groupingId, mdb_collection, rlvcStats_collection, recEngineInputLimit=100000*)

Creates a MongoDB table with the relevancy score statistics of the documents passed into the current preProcess run

Arguments:

topNPercent : *float*

the top n percent of documents with the highest relevancy scores that will be queried for possible recommendation by **submission.py**

groupingId : *string*

the grouping Id for which the *recConfig* collection contains a particular document of configurables

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

rlvcStats_collection : *mongodb.collection*

a MongoDB collection pointer in which relevancy score statistics for given **groupingId**s are stored

recEngineInputLimit : *int (optional)*

the limit of the number of documents that can be queried for possible recommendation by **submission.py**

Returns:

Inserts a new or updates an existing document in the *rlvcStats* collection with relevancy score statistics for the given **groupingId**

preProcess.run_preProcessing(*groupingId, mdb_collection, rlvcStats_collection, recConfig_collection, sw_set, recConfig_handle*)

Configures setup variables and runs the preProcessing/Relevancy Engine

Arguments:

groupingId : *string*

the grouping Id for which the *recConfig* collection contains a particular document of configurables

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

rlvcStats_collection : *mongodb.collection*

a MongoDB collection pointer in which relevancy score statistics for given **groupingId**s are stored

recConfig_collection : *mongodb.collection*

a MongoDB collection pointer in which configuration parameters for **preProcess.py** and **submission.py** for given **groupingId**s are stored

sw_set : *set*

a set of stopwords to be ignored when tokenizing

recConfig_handle : *mongoBall.recConfigURI*

a handle on a *recConfig* document in the *recConfig* collection and its parameters for the given **groupingId**

Returns:

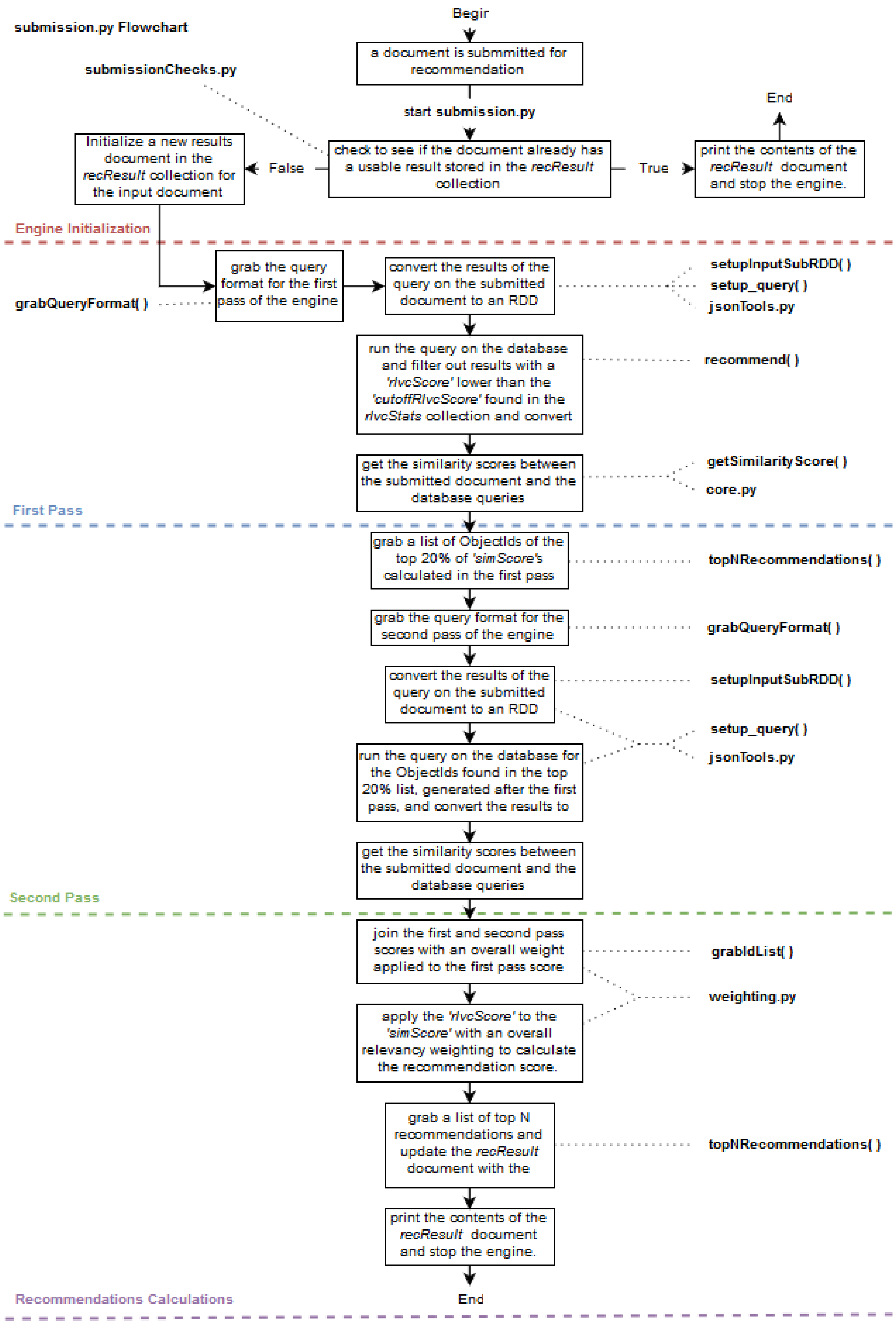
Inserts a new 'rlvcScore' field or updates an existing one with the relevancy score generated by the preProcessing/Relevancy Engine for all documents passed into the engine

Notes on the Relevancy Engine

Weighting Algorithms:

Explanation of which algorithms are currently being used for the weighting process, and why these algorithms have been chosen for certain fields.

Recommendation Engine: submission.py


submission.py Flowchart
Begir
a document is submmitted for recommendation
start submission.py
submissionChecks.py
check to see if the document already has a usable result stored in the recResult collection
False
Initialize a new results document in the recResult collection for the input document
True
print the contents of the recResult document and stop the engine.
End
Engine Initialization
grab the query format for the first pass of the engine
grabQueryFormat( )
convert the results of the query on the submitted document to an RDD
setupInputSubRDD( )
setup_query( )
jsonTools.py
run the query on the database and filter out results with a 'rlvcScore' lower than the 'cutoffRlvcScore' found in the rlvcStats collection and convert
recommend( )
get the similarity scores between the submitted document and the database queries
getSimilarityScore( )
core.py
First Pass
grab a list of ObjectIds of the top 20% of 'simScore's calculated in the first pass
topNRecommendations( )
grab the query format for the second pass of the engine
grabQueryFormat( )
convert the results of the query on the submitted document to an RDD
setupInputSubRDD( )
setup_query( )
jsonTools.py
run the query on the database for the ObjectIds found in the top 20% list, generated after the first pass, and convert the results to
get the similarity scores between the submitted document and the database queries
Second Pass
join the first and second pass scores with an overall weight applied to the first pass score
grabIdList( )
weighting.py
apply the 'rlvcScore' to the 'simScore' with an overall relevancy weighting to calculate the recommendation score.
grab a list of top N recommendations and update the recResult document with the
topNRecommendations( )
print the contents of the recResult document and stop the engine.
End
Recommendations Calculations

submission.py is the core application that is submitted to spark-submit for the recommendation engine. This application is called in the following way:

```
~$ submission.py submissionType inputSub <mongoDB_uri>
                 <groupingId> <mongoDB_collection>
                 <results_collection> <rlvcStats_collection>
                 <message_type> <weightedBool>
                 <recConfig_collection> <recConfig_uri>
```

Arguments:

submission_type : *string*

the type of input submission. This will be either ”file”, ”single”, or ”query”

inputSub : *object*

documents that recommendations are being requested for. This will be either a JSON dictionary, a file name, or a MongoDB query

mongoDB_uri : *string (optional)*

URI of the Mongo database that contains the documents to be recommended against (default: ”mongodb://user:pass@localhost:27017/test”)

groupingId : *string (optional)*

the grouping Id that the input documents belong to in the Mongo database (default: ’default’)

mongoDB_collection : *string (optional)*

name of the MongoDB collection that contains the documents that will be recommended (default: ”message”)

results_collection : *string (optional)*

name of the MongoDB collection that contains the results of the recommendation that will be recommended (default: ”recResult”)

rlvcStats_collection : *string (optional)*

name of the MongoDB collection that contains the statistics of the ’rlvcScore’s of the given **groupingId** (default: ”rlvcStats”)

mdb_collectionType : *string (optional)*

type of messages that are contained in the MongoDB collection (default: ”incidents”)

weightedRlvc : *boolean (optional)*

whether or not the recommendation engine will also take into account the ’rlvcScore’ of the documents when calculating the recommendation score (default: True)

recConfig_collection : *string (optional)*

name of the MongoDB collection that contains user-configurable variables for use in both the recommendation engine and the pre-processing engine (default: ”recConfig”)

recConfig_uri : *string (optional)*

URI of the Mongo database that contains the user-configurable variables documents (default: same as **mongoDB_uri**)

submission.**grabQueryFormat**(*mdb_collectionType=”incidents”, firstPass=True*)

Determines the query format that the recommendation engine will use to grab documents from MongoDB based on the collection/ticket type

Arguments:

mdb_collectionType : *string (optional)*

the type of ticket being searched and recommended against

firstPass : *boolean (optional)*

flag for whether the engine is on its first or second pass

Returns:

query_format : *dictionary*

format of the mongo query for a specified ticket type

submission.**setup_query**(*idList, mdb_collection, query_format, sc*)

Given a list of MongoDB ObjectId’s and a query format, this function fetches the results of the queries and concatenates their values into a single string field and parallelizes that string field into a Spark RDD

Arguments:

idList : *list*

a list of ObjectIds corresponding to the documents in the collection

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

query_format : *dictionary*

format of the mongo query

sc : *SparkContext*

Spark Context Environment

Returns:

subRDD : *spark.RDD*

a concatenated Spark RDD of the submitted documents

submission.**recommend**(*mdb_collection, rlvcStats_collection, groupingId, query_format, sc*)

Sets up the query corpus of the dataset to be recommended against

Arguments:

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

rlvcStats_collection : *mongodb.collection*

a MongoDB collection pointer in which relevancy score statistics for given **groupingId**s are stored

groupingId : *string*

the grouping Id that the input documents belong to in the Mongo database

query_format : *dictionary*

format of the mongo query

sc : *SparkContext*

Spark Context Environment

Returns:

queryRDD : *spark.RDD*

a concatenated Spark RDD of the query corpus of the dataset to be recommended against

submission.**getSimilarityScore**(*subRDD, queryRDD, sw_set, sc*)

Calculates the similarity score for the documents submitted to the engine against a set of queried documents

Arguments:

subRDD : *spark.RDD*

a concatenated Spark RDD of the submitted documents

queryRDD : *spark.RDD*

a concatenated Spark RDD of the query corpus of the dataset to be recommended against

sw_set : *set*

a set of stopwords to be ignored when tokenizing

sc : *SparkContext*

Spark Context Environment

Returns:

similaritiesRDD : *spark.RDD*

a Spark RDD of cosine similarities between the submitted documents and the query documents

recommendations : *list*

an accending 'simScore'-sorted list of similar documents found for the submitted documents

submission.**topNRecommendations**(*recommendations, n, inputSub*)

Grabs a list of the top n recommendations found for given input documents

Arguments:

recommendations : *list*

an accending 'simScore'-sorted list of similar documents found for the submitted documents

n : *int*

the number of top recommendations to be returned

inputSub : *object*

input documents that recommendations are being requested for

Returns:

resultsList : *list*

a descending sorted list of the top n recommendations and their similarity scores of the form: (ObjectId,'simScore')

submission.**grabIdList**(*recommendations*)

Grabs a list of ObjectIds from a list of recommendations while maintaining the 'simScore'-sorted order

Arguments:

recommendations : *list*

an accending 'simScore'-sorted list of similar documents found for the submitted documents

Returns:

recIdList : *list*

a list of ObjectIds of the returned recommendations in accending order of 'simScore's associated with the ObjectIds

submission.setupInputSubRDD(*inputSub, mdb_collection, query_format,sc, submission_type*)

Sets up the a Spark RDD for the input ObjectId or list of input ObjectIds

Arguments:

inputSub : *object*

input documents that recommendations are being requested for

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

query_format : *dictionary*

format of the mongo query

sc : *SparkContext*

Spark Context Environment

submission_type : *string*

the type of input submission. This will be either ”file”, ”single”, or ”query”

Returns:

subRDD : *spark.RDD*

a concatenated Spark RDD of the submitted documents

insertedIds : *list*

a list of ObjectIds of any documents that were inserted into **mdb_collection** (this value is ’None’ for a **submission_type** of ’query’) submission.**run_engine**(*submission_type, inputSub, groupingId, mdb_collection, mdb_collectionType, sw_set, sc, mdb_results_collection, rlvcStats_collection, recConfig_handle, weighted=True*)

Configures setup variables based on the submission type and runs the Recommendation Engine

Arguments:

submission_type : *string*

the type of input submission. This will be either ”file”, ”single”, or ”query”

inputSub : *object*

input documents that recommendations are being requested for

groupingId : *string*

the grouping Id that the input documents belong to in the Mongo database

mdb_collection : *mongodb.collection*

a MongoDB collection pointer

mdb_collectionType : *string (optional)*

the type of ticket being searched and recommended against

sw_set : *set*

a set of stopwords to be ignored when tokenizing

sc : *SparkContext*

Spark Context Environment

mdb_results_collection : *mongodb.collection*

a mongoDB collection pointer to the *recResult* collection

rlvcStats_collection : *mongodb.collection*

a MongoDB collection pointer in which relevancy score statistics for given **groupingId**s are stored

recConfig_handle : *mongoBall.recConfigURI*

a handle on a *recConfig* document in the *recConfig* collection and its parameters for the given **groupingId**

weighted :

Returns:

Inserts a new document or updates an existing document in the *recResult* collection with the top recommendations returned by the engine for the given **groupingId**.

Each successfully completed document in the *recResults* collection has the following layout:

```
{   "_id" : ObjectId("573b8f6b395dfedd3a9fac18"),
    "inputId" : ObjectId("573b8e5d7f9ee95d1f24b702"),
    "status" : 3,
    "groupingId" : "538f9e119e16abd715913ab5",
    "startTime" : datetime.datetime(2016, 6, 3, 4, 15),
    "output" : [
        { "outputId" : ObjectId("573b8e0f7f9ee95d1f249734"), "score" : 0.528232112139983 },
        { "outputId" : ObjectId("573b8e5c7f9ee95d1f24b6d3"), "score" : 0.20892430567720943 },
        { "outputId" : ObjectId("573b8e5c7f9ee95d1f24b6c2"), "score" : 0.1532693624738012 },
        { "outputId" : ObjectId("573b8e5d7f9ee95d1f24b70b"), "score" : 0.047276274163356 },
        { "outputId" : ObjectId("573b8e5c7f9ee95d1f24b6d8"), "score" : 0.040656897402438684 } ],
    "endTime" : datetime.datetime(2016, 6, 3, 4, 32),
    "result" : 1}
```

Notes on the Recommendation Engine

The Two-Pass System:

Explanation of the two-pass system and what combinations of field queries for the first and second passes have already failed in past tests. How the similarity score is calculated.

Relevancy Weighting:

Explanation of how the relevancy score is taken into account when calculating the recommendation score.