UC1ST1102 - Noodle - Project documentation

Introduction

The purpose of this document is to describe the functionality in this project at a high level. As there are a lot of components, this document will be formatted as such:

Introduction (this).
Top-level narrative of main functionalities.
Continued top-level narrative of main functionalities.
High-level interfaces.
All modules.

Parlance (commonly used):

DataObject/DataObj:
: This is simply a container which is used throughout this project. Data contained is tweet data, additions to tweet data, as well as alterations.

Siminet/Similarity net:
: This is a data structure which is usually stored inside a list. The data structure is a tree which is made of similar words and some confidence score. Siminets will be described in greater detail in the next section of this document.

Ring (in the context of DB):
: This will refer to components of the database structure. They are essentially tree branches which loop back against their bases, effectively making graphs. The purpose is related to implementation and analysis simplicity. Rings will be discussed in greater detail in the section named “Continued top-level narrative of main functionalities”.

Twitter noodle:
: This will refer to this project, all its components and intent.

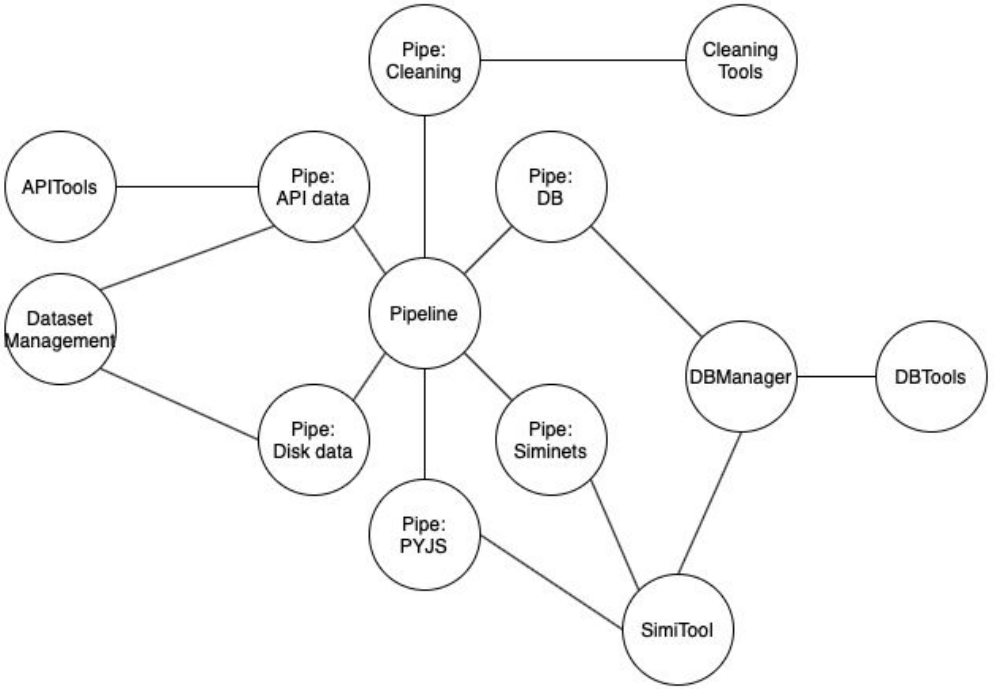
Pipe:
: Top-level interface which hides away implementation. Conceptually, a pipe pulls data from somewhere, processes it and sends it somewhere else. The manifestation of pipes is subclasses of 'PipeBase', usually collected in a 'Pipeline' class which does a specific sequence of processes. This will be explained more further down.

A top-level narrative of a pipeline.

The aim of twitter noodle is to pull tweets from Twitter or local storage, clean them, do similarity assessment and put them either in a Neo4j database or a front-end where some information is plotted on a chart in real-time. The reason for doing these processes is to be able to detect 'incidents', display some measuring output and store everything in a performant structure. 'Similarity assessment' is fundamentally performed by creating siminets and checking them against other siminets to get a score of total similarity. The next few paragraphs will attempt to explain this by illustrating a pipeline, the format will be as such:

- Introduction of an illustration which will be used as a visual reference point.

- Conceptual narrative which will describe a pipeline used to pull tweets from twitter, clean them, add siminets and push them to a front-end. Specific implementation details and tools will be mostly avoided in this part.

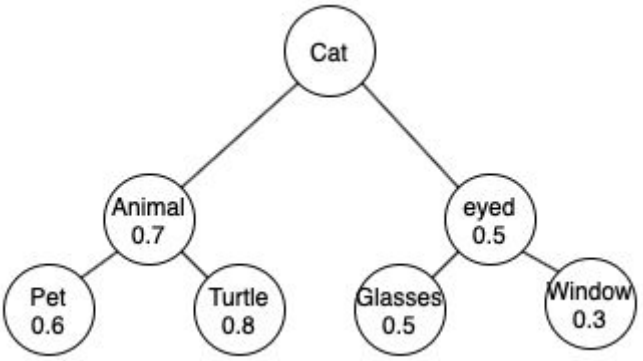


This illustration is one way of viewing twitter noodle, some nodes are occasionally connected, even though it is not explicitly illustrated here.

Firstly, twitter data is pulled with the 'API pipe'. As this example does not concern datasets already stored on local storage, the components 'Dataset management' and 'Disk pipe' will naturally be ignored at this stage.

The tweet data pulled with the 'API pipe' is then moved (one-by-one) to the 'Cleaning pipe', where tweets are converted into data objects with some specific fields. After the conversion, all text fields (main body of tweets) are 'cleaned', meaning that everything without any concrete use (for example stopwords and/or links) is either removed or filtered away into dataobj properties. A sentiment score is calculated at this point to get an approximation of the nature of the dataobj; this was intended to be used for making incident detection and similarity assessment more precise, though was never really implemented due to time constraints.

By now, the data is cleaned, formatted and can be passed along to 'Siminet pipe'. This pipe uses a 'SimiTool' which essentially leverages a word2vector model to create a tree structure of similar words. Siminets are very similar to word-nets but are automatically generated and more concerned with relationships between words in a twitter context (model is based on twitter data).



This is a very simplified siminet, normally the difference between words is more subtle and the total node count is usually a lot higher.

A siminet is created for each word in the text field of a dataobject and the total node count can vary. Generally, the theoretical node count is:

$$\sum_{i>1}^{1} x\hat{}i$$

In this expression, '*x'* represents the number of child branches for each word, which is 10 by default, while '*i'* is the total amount of tree levels. So if there are 10 branches for each word, and there are 2 levels, then the node count is approximately 110. This number can vary, as stated above, for three reasons:

1: Word2vec model can be incomplete and may not recognise certain words, thus cannot branch.
2: Bad branch content. For instance, words with only one character are not allowed to branch.
3: Duplicate words across the tree are removed during the process. In the illustrated example above, 'Cat' has the child branch 'Animal', and 'Animal' can occasionally get a child branch 'Cat'. This would give two or more instances of 'Cat', which is redundant. To fix this problem, some compression is done such that duplicates are removed, without reducing the total similarity score.

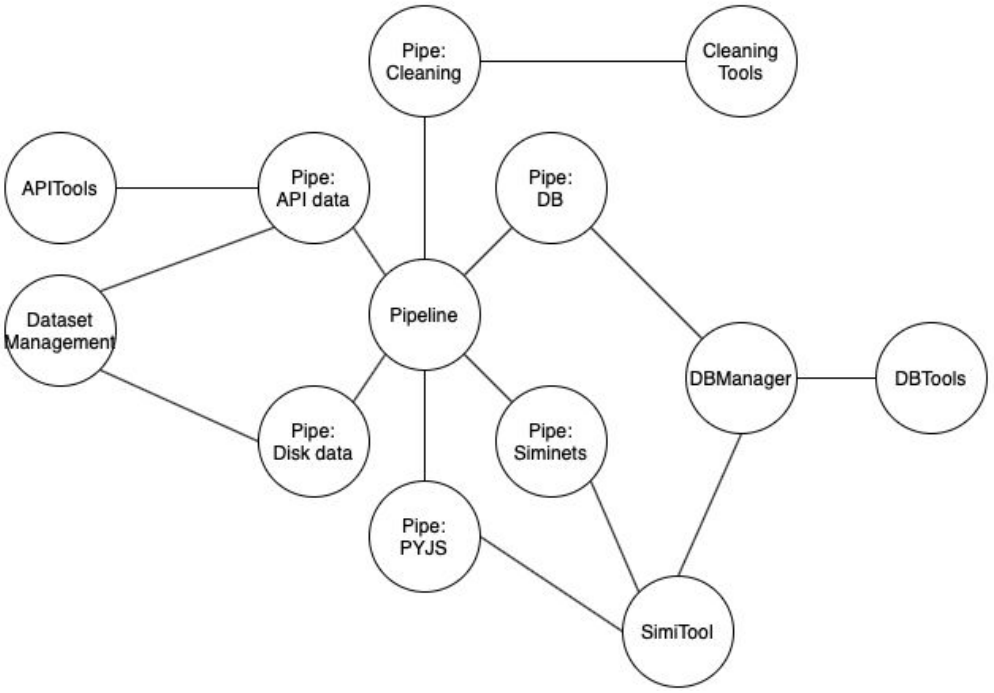
After a dataobject has gained a siminet, it is pushed along to the final pipe where it can be used for visualisation on a front-end. The PYJS bridge pipe is set ut such that it can get a query which is automatically transformed into a siminet. This query siminet is then checked against the siminets of all incoming dataobjects, which results in a series of confidence scores. As this is done, the results are pushed over WebSockets to a front-end for visualisation purposes:

Continued top-level narrative of main functionalities

In the previous section, the demonstrated pipeline fetched data from the Twitter API, cleaned and transformed it, added siminets, and used that to send some values to the front-end based on a query. At the moment of writing, there are 4 types of pipelines which consist of 4 types of pipes each:

- 1: API + Cleaning + Similarity + PYJS
- 2: API + Cleaning + Similarity + DB
- 3: Disk + Cleaning + Similarity + PYJS
- 4: Disk + Cleaning + Similarity + DB

This section of the documentation will focus on pipeline number 4 because it introduces the 'disk pipe' (local storage) and DB pipe. The cleaning and similarity pipes will not be discussed too much because they have already been described with some detail in the previous section.



Adding illustration again for reference.

Before using this pipeline (number 4), a dataset needs to be available. This is done by using 'Dataset Management' tools, as is visualised in the illustration above. These tools have two main purposes:

- Generation a dataset.
- Scaling a dataset (splitting and merging).

The dataset generator uses the 'API pipe' to fetch data and stores it in a predefined amount of chunks for some memory management control. The scaling tool can be used if the dataset sizes are for some reason inconvenient.

The first pipe, 'Disk pipe', simply streams data from a locally stored dataset and makes it available for other connected pipes.

The second and third pipe, used for cleaning data and adding siminets, are attached as well. These pipes were discussed in the previous section of this document so going into detail again is redundant at this stage.

The final pipe is simple by itself but uses a lot of complicated underlying tools for database(Neo4j) insertion and sorting. The purpose of using a graph database is to be able to cluster similar dataobjects together and retrieve data efficiently. The remaining of this section (Continued top-level narrative of main functionalities) will explain what the DB pipe does in practice.

There are three types of relationships used in the database: TICK, UP and DOWN. First, each individual relationship type will be described. After that, some illustrations will be shown as a realistic example.

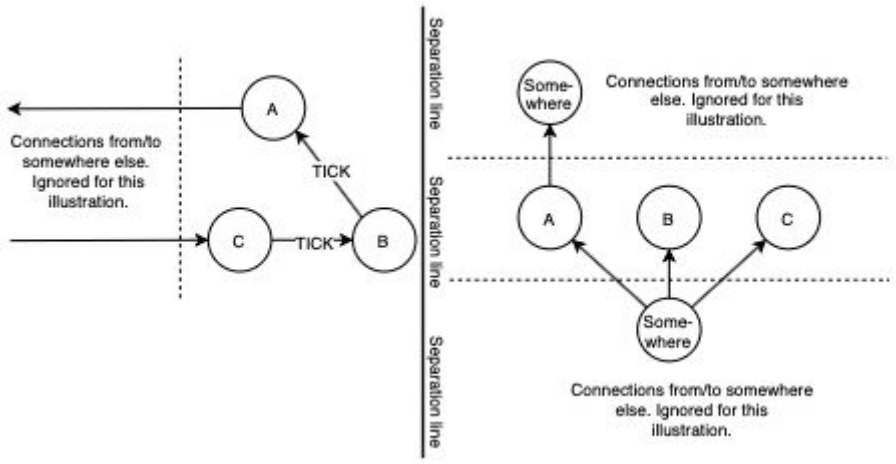


Illustration for TICK relationship. Each side of the separation line is a different representation of one identical relationship. The left side represents a graph, while the right represents a tree.

TICK can be be viewed as a sibling relationship. Using the illustration above, the nodes in the example on the left side of the separation line, node A, B and C are all related to each other by their parent. This can be illustrated with the tree example on the right side of the separation line. Nodes which are in a sequence of TICK relationships are referred to as 'rings' in this project.

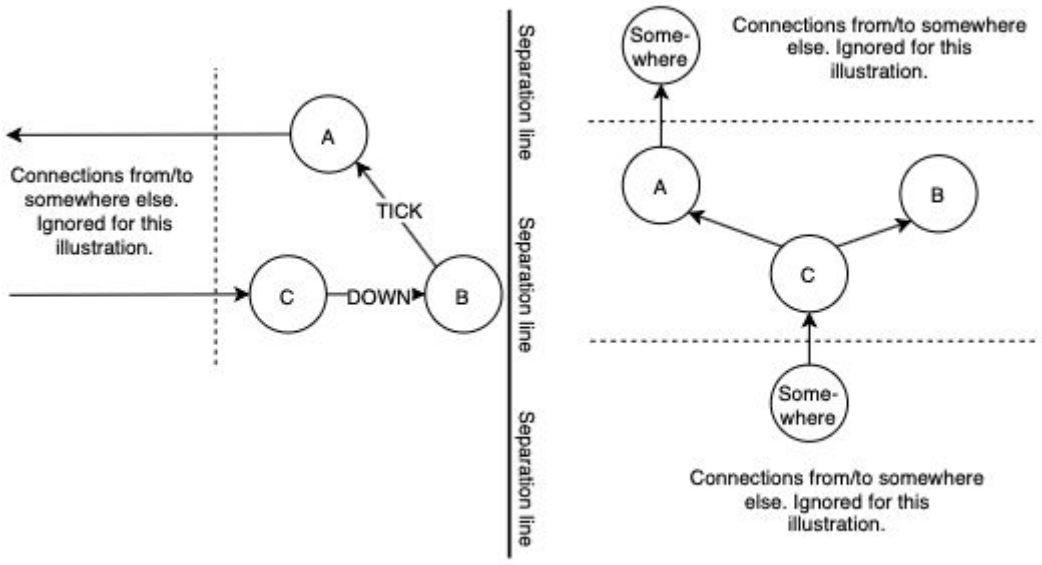


Illustration for DOWN relationship type. Each side of the separation line is a different representation of one identical relationship. The left side represents a graph, while the right represents a tree.

DOWN is specifying a parent/child relationship. In the illustration above, node B has an incoming DOWN from C and an outgoing TICK to A, meaning that A and B are on the same 'ring' and are both child nodes of C. The tree example on the right side of the separation line demonstrates this. The main function of DOWN is to mark where a new ring is created and what the first item in that ring is. This is done for computational/implementation reasons and means that, if the ring consisting of node A and node B is represented as a tuple, then node B comes before node A.

The UP relationship is almost identical to DOWN. Using the illustration above (left side): If node A has an outgoing UP relationship, then it is known that node A is the last node on its ring.

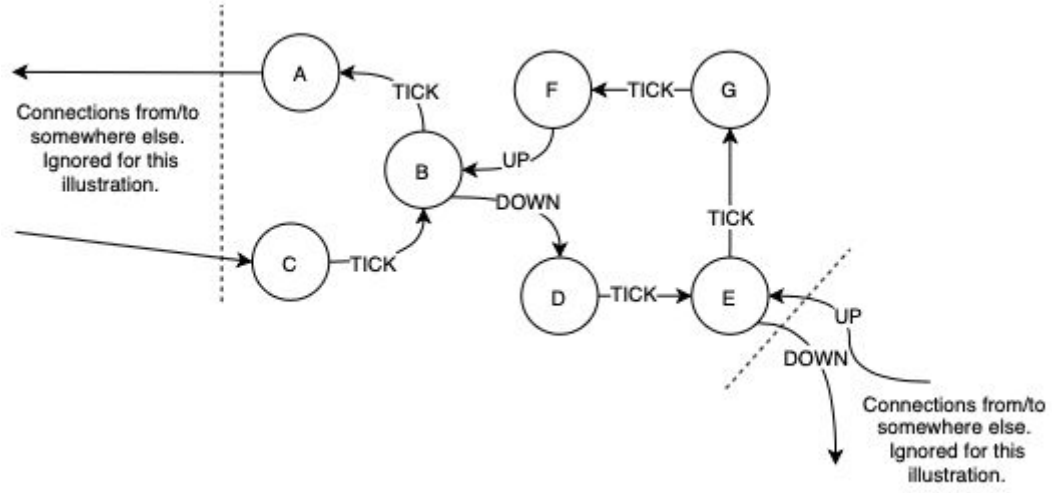
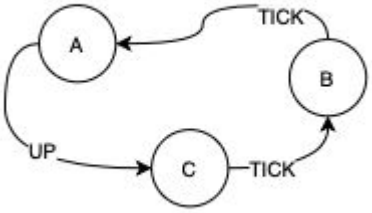


Illustration for all relationship types.

All relationship types combined makes it simple to analyse what is going on in a graph. A, B and C are all on the same ring which has an unknown continuation (hidden for simplicity). B is seen as a parent,

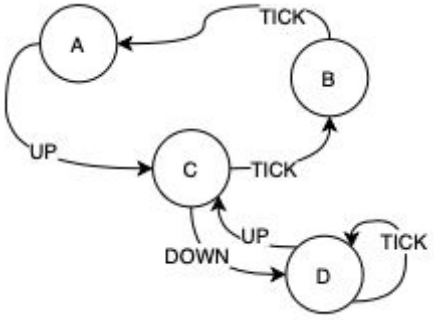
with a child/sub ring consisting of D, E, F, G. Likewise, E has its own child/sub ring, though the exact details are cut off for simplicity purposes here as well.

Having illustrated the different relationship types and how the schema is created, it is now easier to continuing the pipeline example, with the last pipe being the DB pipe, and demonstrating the steps involved in doing DB insertion. Firstly, an initial/root ring must exist (one will automatically be created if that is not the case). Each DB node stores all the information in a dataobject and the relationship types are based on siminet comparisons:

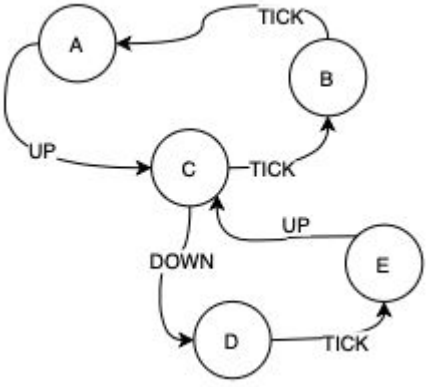


This could be an initial ring with level0, meaning that it is a root ring. All other rings have a level > 0. A thing of note in this illustration is that C has an incoming UP relationship from A. This is the only deviation from the schema rules and was allowed such that it is possible to know what the first and lasts nodes are in the entire database structure.

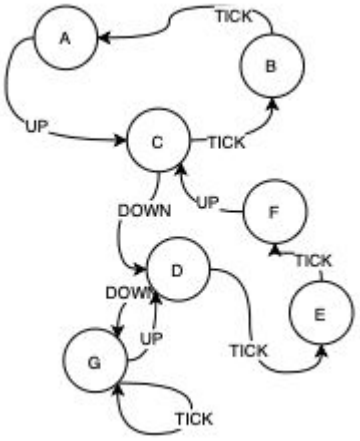
A new node (D) is introduced to the database. All nodes on the root ring are fetched and their siminets are compared to the siminet of D. The comparisons will result in an output pointing to the node in the ring which is most similar to the new node; in this case C. This creates a new ring beneath C, which only consists of D.



In the next example, a new node (E) is introduced. It is most similar to C on the root ring and it is known that E should go beneath C. However, there is an arbitrary minimum ring size(3), so E will just be appended to the ring which D is part of.

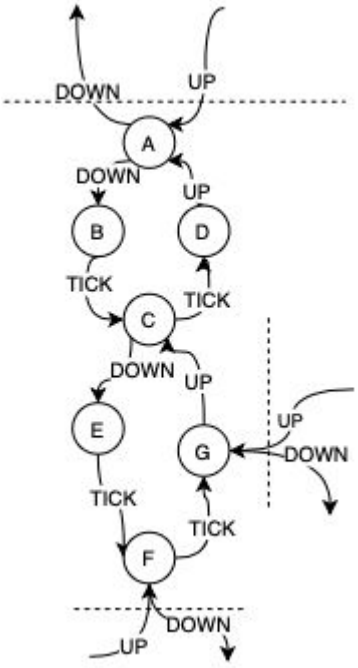


In the next image on the right, two things happened:
Node F was inserted first; much like E, it was just placed at the end of the ring to fill a minimum ring size. After F was inserted, the node G was introduced. It was compared against the root ring and was coincidentally also most similar to C. As the subring of C had a minimum ring size requirement filled, that ring could also be used for similarity comparisons when introducing G. Out of the ring containing D, F and E, the node marked with D was most similar to G.
This made a new ring under D which only contains G.
If G was not similar to any of the nodes on the ring below C, then it would just be appended at the end of that ring.



This process of introducing new nodes and comparing them against rings is done recursively. Root ring is always first, then a subring of a node in the root ring, etc. Each ring effectively represents clusters and sub-clusters, which separates topics and sub-topics from each other. The benefit of doing this is mainly to have the ability to do topic modelling and storing everything efficiently such that topic retrieval is performant. Using a graph structure makes it unnecessary to check all the data, only a subset.

This DB insertion procedure is naturally flawed, though. Firstly, it is very dependent on the underlying word2vec model and similarity calculation, meaning that topics can be misidentified and placed somewhere incorrectly. Secondly, a similarity 'drift' is almost guaranteed to occur: If a new node is inserted, there might be a future ring which is more suited for it than the ring it was originally placed on. This makes it necessary to do periodic sorting of the entire structure (in twitter noodle, this is done incrementally).



For instance, a new node (N) is introduced and compared to all nodes on the ring containing A, B, C and D. It is not similar enough to any of them and is appended to that ring. But what if it was similar to F and should have been placed on its subring? That is a very realistic example and there is initially no way of mitigating this flaw.

The approach to this issue is to use database sorting, as mentioned further up. Continuing this example, the database sorting algorithm can look at the ring which F is a part, including the parent (C) and sort the ring such that there is a specific ordering. The order is 'most representative' to 'least representative'. If a node is 'most representative', then there is a strong similarity link between it and all other nodes on that ring. This increases the likelihood that it can represent the topic/cluster it is a part of and catch new nodes easier.

In practice, F and C can swap places through sorting (F could also be moved to the root ring through this process). Continuing the example, the new node N is not similar at all to A, B, C and D, but is similar to F. If sorting is been done and F and C have swapped places, then N is inserted more meaningfully beneath F, instead of just being added to the end of the ring containing A.

High-level interfaces

In the previous two sections, examples were given for how pipes work conceptually; this section is written for a more detailed perspective of what they actually are. Pipes are simply top-level interface objects which are combined through a pipeline object for simple usage and automation. There are 6 types of pipes in twitter noodle, excluding the base class (PipeBase) :

- CleaningPipe
- DBPipe
- FeedFromAPIPipe
- FeedFromDiskPipe
- PyJSBridgePipe
- SimiPipe

Each pipe can be connected to each other by using the property 'previous_pipe', which is set as a reference to a pipe on init. For example, an instance of 'FeedFromAPIPipe' can be set as 'previous_pipe' in a new instance of CleaningPipe.

When a pipe is set up, it can be used by calling the 'process' method. This method is uniform and is attached through the base class. It will generally attempt to access the output property (list) of a 'previous_pipe'; an item will be popped, processed and appended to the output list of the current pipe. The processing will vary, depending on what pipe is doing it, but will generally follow this structure:

- task_method (x) // Current pipe class
    - Do something specific to the pipe, usually converting x to y.
    - Optionally return y.
- process_method(): // Base pipe class
    - Attempt to access data from previous_pipe
    - Pass some data to task_method, defined during init.
    - If processed, add to output property

This section will be a brief overview of what each pipe does and what modules they use:

- CleaningPipe expects a 'previous_pipe' which has 'tweepy tweet' objects in its output list. Data is pulled, transformed into dataobjects and cleaned. Using:
    - packages.cleaning.data_object_tools
    - packages.cleaning.basic_cleaner

- DBPipe expects a previous_pipe which has an output list containing dataobjects with siminets. Data is pulled and inserted into the database. Additionally, database sorting is called from this pipe. Using:
    - packages.db.db_mana (DB manager, which uses db_tools from same directory)

- FeedFromAPIPipe does not expect a previous pipe; it uses "tweepy" to access Twitter API. The 'process' method of this class does not need to be called because tweets are sent to the output property (of this class) on a separate thread handled by 'tweepy'. However, calling 'process' is allowed because it reduces the need for special cases: if this pipe is collected with other pipes, then iterating over the collection and calling 'process' is simple. Using:
    - packages.feed.tweet_feed

- FeedFromDiskPipe does not expect a previous_pipe. It accepts a path to a dataset file which is used to stream data(using a generator) to an output list when calling 'process'. Using:
    - Only standard library

- PYJSBridge expects a previous_pipe which has an output list containing dataobjects with siminets attached. This pipe accepts queries which it uses to create new siminets, and on each 'process' call, these siminets are compared against the siminets of incoming dataobjects to create similarity scores (sent over WebSockets). One thing to note is that this pipe requires a similarity tool to create siminets from queries; it attempts to go through all previous pipes to find one. If this fails, a new similarity tool is instantiated. Additionally, this pipe spawns a thread the first time 'process' is called, to start a WebSockets server. Uses:
    - asyncio, WebSockets, threading
    - package.similarity.process_tools
    - credentials

- SimiPipe expects a previous_pipe with an output list containing dataobjects. This pipe instantiates a siminet tool and uses it to attach a siminet on all incoming dataobjects each time 'process' is called. Uses:
    - packages.similarity.process_tools

All modules

This section will list all modules and components in the order they appear in the project hierarchy. Some sections will address one module at a time, while others may describe a whole folder, depending on what makes the most sense. All functions will be listed, though the detail level of each description will be low because the use case of this section is to provide a reference for function calls while reading the code (which itself contains detailed doc-strings). The format of this section will generally be as such (though some deviation will occur):

Module/folder/filename.
Path.
Description.
Functions/methods:
function1_name: short description and purpose.
function2_name: …
…
…

credentials (module)
Path: Project root folder.
Description: Stores credentials for API and the local webserver.

db_query (script)
Path: Project root folder.
Description: Simple CLI for querying the Neo4j DB.
Functions:
get_db_tool()

returns an instance of a tool used to interface the DB

set_query()

Accepts a previous query and a DB tool (instance). This function asks for user input, converts it to a siminet (using the DB tool) and updates the previous query given as a parameter.

display_data()

Accepts a list of dataobjects which the function prints out in a tidy format.

main()

This is a light-weight menu system for getting queries (using set_query()) and printing them out (using display_data())

front-end (directory)

Path: Project root folder.

Description: This folder contains a HTML/CSS/JS combination used for receiving data from a Python Websockets server and visualising it as a real-time chart. The visualisation is built with the p5.js library.

- **index.html** ties all other files together and points to an external source for the p5.js library.
- **style.css** contains some bare-bones markup.
- **client.js** contains a function which is used to start listening with Websocket.
- **sketch.js** is the main front-end. It contains the following functions:
  - **setup()** Calls the function in client.js and initiates p5.js canvas.
  - **draw()** is the main loop used by p5.js for drawing, it uses timeDilation() to get values sent through Websockets, lerps the values for smooth transitions, and calls renderWave() for the actual drawing.
  - **TimeDilation()** is used to zoom in and out of the visualised chart, using the global 'dilation' variable as a factor.
  - **renderWave()** uses the values obtained from the Python backend to draw a chart.

  This file also contains a ColorPoint class which is used for colourisation in renderWave().

module_installer (script)

Path: Project root folder.

Description: This script only contains a function for installing all required python modules in this project.

basic_cleaner (module)

Path: root/packages/cleaning/basic_cleaner.

Description: This is a static class used for NLP text cleaning.

Methods:

- **print_comparison()** is called from autocleaner() to see what changes have been made to a dataobject after cleaning. It essentially only does a printout of a dataobject before and after cleaning.
- **autocleaner()** takes in a dataobject and 'cleans' its text fields by doing a sequence of calls to the other method in this class. Also accepts a sentiment range, which must be a list of two floats which represent a range (example: [-1.0,1.0]). This is used to gauge the overall sentiment in the dataobject.
- **remove_duplica_words()** takes in a string, removes words which appear more than once and returns the result.

- **clean_stopwords()** accepts a string, removes all stop-words and returns the result. Stop-words are retrieved from an external module(custom_stopwords).
- **clean_punctuation()** accepts a string, removes all non-alpha characters and returns the result.
- **clean_links()** accepts a string, removes all perceived links using regular expression and returns the result.
- **clean_hashtags()** accepts a string and filters out hashtagged words such as '#pineapplepizza'. The return is a list where index 0 is the text without hashtags, while index 1 is the filtered hashtag.
- **clean_alphatabs()** accepts a string and filters out alphatagged such as "@therealporcupine". The return is a list where index 0 is the text without alphatags, while index 1 is the filtered alphatag.
- **clean_convert_to_lowercase()** accepts a string, converts all characters to lower-case and returns the result.
- **set_sentiment()** accepts a string and a list representing a range. The range must be a list of exactly two floats, first for the lower range and second for the upper range. The argument string is passed to TextBlob (external module) to get a sentiment score which is compared to the range(param). If the sentiment is within the range, a True is returned, else False.

custom_stopwords (module)

Path: root/packages/cleaning/custom_stopwords

Description: This module fetches stopwords from nltk.corpus (external module) and combines them with a custom stopword list.

Functions:

- **__file_to_list()** accepts a path to a file containing stopwords and a list which is used to add the file content (in-place).
- **main()** uses __file_to_list() and nltk to create and return one set of stopwords.

data_object_tools (module)

Path: root/packages/cleaning/data_object_tools

Description: This module contains some functions related to dataobject creation and field transformation.

Functions:

- **convert_tweet2dataobj()** accepts a 'tweepy tweet' type, converts it into a dataobject and returns the result.
- **siminet_to_text()** takes in a siminet, a row delimiter and a column delimiter. This is used to convert a siminet(2d list) into a string, using the delimiters specified as arguments. Returns the siminet string.
- **txt_to_siminet()** reverses the process of siminet_to_text(): Takes in a string which represents a siminet and uses the delimiter arguments (row_sep & col_sep) to convert it to a 2d list. Returns the siminet list.

data_object (module)

Path: root/packages/cleaning/data_object

Description: This module contains the DataObj class. It is a class containing some parameters. This class is used throughout twitter noodle to pass around tweets.

dataset_tools (directory)

Path: root/packages/dataset_tools

Description: This Directory contains three files used for managing datasets.
The file named 'common' contains support functions for the other two files, 'generate_dataset' contains tools for creation datasets, while 'scale_dataset' contains tools for merging and splitting datasets.

Functions in **'common'**:

- **reformat_tweet_datetime()** expects a tweepy tweet and accesses its 'created_at' field. The value in this field is then converted to a certain format: YYMMDD-HH_MM_SS. The function returns the formatted value as a string.
- **get_new_filename()** accepts a list containing tweepy tweets and a string which points to an output directory. The first and last tweet in the list are passed to reformat_tweet_datetime() to create a time range which represents the list. This time range is combined with the output directory (argument) and returned.
- **save_data()** expects a list of tweepy tweets, a string specifying the output directory, a bool specifying whether or not the saved data should be compressed, and another bool specifying if the function should print the output path. This function spawns a thread for saving the data with pickle such that it is non-blocking (useful for timing precision). As such, the input list is copied in case the content is deleted before the thread is done.

Functions in **'generate_dataset'** (module with class):

- **init()** makes sure some essential information is given: runtime_total, runtime_between_slices, runtime_forever, out_directory, track_keywords and compression.
  'runtime_between_slices' specifies how many seconds should pass between each save. For example, if runtime_total is 10 and runtime_between_slices is 5, then the script will generate two files.
  'track_keywords' is used to tell Twitter API which tweets to provide.
- **print_progress_bar()** is used to print a CLI progress bar.
- **validate_session()** is used to validate that all instance properties of self are set.
- **get_time_left()** checks how much time is left before the run_collector() is done.
- **run_collector()** sets up the data gathering tool and uses a loop to keep track of time, save data and print progress.

Functions in **'scale_dataset'** (module):

- **get_filenames_in_dir()** accepts a string specifying a directory path and returns the names of all files in that directory.
- **get_file_content()** accepts a file name and attempts to load data from it, using bz2 and/or pickle (whichever works). Content is returned.
- **sort_tweets_chronologically()** accepts a list of tweets which is sorted in-place by their timestamps.
- **merge_dataset_by_directory()** accepts an input directory string and an output directory string. Attempts to combine all datasets(made with generate_dataset) in the input directory into one list and writing it out to the output directory.
- **split_dataset_by_obj_count()** accepts a divider integer, a filename for the dataset to split, and a string specifying where the output should go. This function will attempt to split the specified dataset by the divider argument as evenly as possible and save all chunks into the output directory.

db_mana (module containing DBMana class)

Path: root/packages/db/db_mana:

Description: This module contains the DBMana class. The call has three primary functionalities; inserting dataobjects into the neo4j database, sorting the database structure, and sending queries to the database.
Most of the class relies on the db_tools module.

Methods:

- **init()** accepts four arguments. 'verbosity' bool is used for conditional printouts. 'ringsize' int specifies what the minimum ring size is in the structure (see 'Continued top-level narrative of main functionalities' section of this document). 'static_ringsize' is used for node insertion; if this argument is True and a new node is not similar to any objects on a current ring, then the node will be appended to that ring. In the same scenario, if the argument is False, then that node will not be inserted anywhere. 'conservative_swap' is used for database sorting; True means that maximum 2 nodes will be swapped on each ring on each sort action.
- **setup_db_tools()** instantiates the class in db_tools and binds it to self.
- **setup_simi_tools()** instantiates class in process_tools and binds it to self.
- **cond_print()** takes in a string and prints it out if self.verbosity is True.
- **create_initial_ring()** accepts a list of dataobjects which are used to create a root ring in the database (see 'Continued top-level narrative of main functionalities' section of this document). This method call will everything in the database before creating a new ring.
- **autoinsertion()** Accepts a dataobject as an argument and inserts it into the database structure. This method is a container for a recursive function which moves from ring to ring in the DB structure.
- **re_introduce_descendants_of()** accepts a dataobject, finds all descendants of it (all rings below it, recursively) in the DB, caches them, deletes them from the DB structure, and re-introduces them from the cache using autoinsertion().
- **clockwork_traversal()** takes two arguments; "sort" bool and 'continuous' bool. This method returns a generator which goes through the database one chunk(ring) at a time for each next() call. If 'sort' argument is True, then sorting will be done for each chunk (using clockwork_sort()). 'continuous' specifies whether or not the generator is infinite: False makes the forementioned generator depleted after all rings are iterated over exactly once, while True makes the generator infinite.
- **clockwork_sort()** takes in a ring (list of dataobjects) and sorts it in the database structure. See 'Continued top-level narrative of main functionalities' section of this document for more a more detailed explanation for what this does.
- **query_auto()** takes a dataobject as an argument and fetches all database rings which are similar to it. Returns a list of dataobjects.
- **query_threshold_selective()** accepts a siminet and a threshold (float). This method traverses the database one ring at a time and compares the similarity score between the argument siminet and the siminets of nodes/dataobjects in the rings. If the similarity scores are greater than or equal to the threshold specified as an argument, then those nodes/dataobjects will be caches. All descendant rings of those cached dataobjects will be checked recursively. All cached dataobjects are returned.
- **query_threshold_all()** accepts a siminet and a threshold. This method traverses the entire database and compares the argument siminet against the siminet of all nodes. If the comparison score is greater than or equal to the threshold argument, then the compared node/dataobject is added to a list which is returned.

db_tools (module)

Path: root/packages.db.db_tools.

Description: This module contains a class which communicates with the neo4j database. The class consists of many methods which send commands to the DB and occasionally return some data.

Methods:

- **init()** Only accepts a verbosity bool, which is used for conditional printout.
- **setup()** uses the config.py file to establish communication with the DB. DB driver is cached to self.
- **print_progress()** uses prints out certain information if self.verbosity is True.
- **delete_all()** deletes everything in the DB.
- **delete_nodes()** accepts a list of dataobjects. These dataobjects are deleted from the database.
- **cache_execute()** iterates over commands in self.cache_commands (list of strings) and executes them (commands are deleted after use). The argument '_single_transaction'=True is used to execute all commands in one chunk but is not recommended (use default False) because it is not properly implemented at the time of writing.
- **execute_return()** accepts a command string which it sends to the database. The return value from the DB is returned from this method.
- **create_tweet_node()** is used for creating nodes in the DB. 'alias' argument is used to name new nodes while they are being created (alias is not used in the DB itself); this is useful if the argument 'mode' is set to 'return' because the alias can be used to add additional information to the returned string. For example;

  A command string is constructed in this function which can be used to create a new node with the alias SOMEALIAS. The command string is returned if the 'mode' argument is 'return'. If the caller wants to add additional information to the node specified in the returned command string, then the node can be referenced with SOMEALIAS.

  The 'obj' param specifies what dataobject should be used to create a new DB node.
  The 'level' is meta information for the ring the new node is going to be attached to; level means how far the root ring is. All nodes on the root ring have 'level'=0.
  The 'mode' param indicates whether the method call will return a command string or append the command string to self.cache_commands.
- **convert_n4jdata_to_dataobjects()** accepts data given by neo4 and attempts to convert it into dataobjects. The result is returned as a list.
- **convert_n4jnode_to_dataobject()** accepts data given by neo4j which represents a singular node and attempts to convert it into a singular dataobject. The result is returned. Note, this is usually called several times from convert_n4jdata_to_dataobjects().
- **get_dataobjects_from_node_by_pkeys()** accepts a list of primary keys which are used to select nodes in the DB and retrieve them as dataobjects.
- **create_initial_ring()** accepts a list of dataobjects which are used to create a root ring in the database structure.
- **get_root_ring()** accesses the root ring of the DB structure and returns it as a list of dataobjects.
- **get_ring_from_obj()** accepts a dataobject, finds the ring it is currently attached to in the DB structure and returns all nodes on that ring as a list of dataobjects.
- **get_level_from_node()** accepts a dataobject and attempts to find out which level it is in. Level in this context refers to how many rings it is away from the root ring. For

example, if the level of a dataobject/node is 0, then it is known that it is in the root ring. The return value is an integer.

- **create_node_endof_ring()** accepts a ring (represented by a list of dataobjects which must all be in the DB structure) and a new dataobject. The new dataobject is added to that ring in the DB.
- **get_last_node_on_ring()** accepts a ring (represented by a list of dataobjects which must all be in the DB structure) and returns the dataobjects which is last, meaning that it has an outgoing UP relationship.
- **create_node_below()** accepts a dataobject which exists in the DB('obj_above') and a new dataobject ('obj_below'). A new ring is created below 'obj_above' and 'obj_below' will be attached to that ring. The DB command is attached to self.cache_commands.
- **get_descendants()** accepts a dataobject and returns all its sub-rings in the DB. Return is a list which can contain dataobjects.
- **get_connectors_of()** accepts a dataobject which exists in the DB and gets all the information about what it is connected to and the relationship types. Returns a list of dictionaries or lists.
- **swap_nodes()** accepts two lists of dataobjects which are used to swap node positions in the DB structure. Example: [a,b] [c,d]: 'a' is swapped with 'c' and 'b' is swapped with 'd'. The instructions are instantly sent to the DB.
- **get_node_next()** accepts a dataobject and a relationship type. Returns value is a list of dataobjects which have the incoming relationship type (arg) from the dataobject specified as an argument.
- **get_node_below()** is a wrapper for get_node_next() where the relationship type is DOWN.
- **get_node_tick()** is a wrapper for get_node_next() where the relationship type is TICK.
- **get_node_above()** is a wrapper for get_node_next() where the relationship type is UP.
- **check_if_last()** accepts a dataobjects and checks if it is the last object/node in the database structure. The return value is a bool.

custom_stream_listener (module)

Path: root/packages/feed/custom_stream_listener.

Description: This module is a container for a class which is used to interface the Twitter API stream provided by tweepy. It is mostly boilerplate.

Methods:

- **init()** accepts a list reference which all new tweet data will go to, a stream toggle, and a verbosity bool.
- **on_status()** is a callback from tweepy which is used to pass new data from the API to the user. The data is added to the output list specified in init if the toggle property is True.
- **on_error()** is a callback from tweepy which is used to handle errors. The implementation toggles of the stream if there is a rate limit warning.
- **out_warn()** is a method which is used to print out things if self.verbosity is True.

tweet_feed (module)

Path: root/packages/feed/tweet_feed

Description: Sets up Twitter API feed with tweepy. Can also load data from local storage.

Methods:

- **init()** sets up tweepy with boilerplate authentication.

- **live_get_listener()** accepts an output list reference which will get all new tweets. The return value is a tweepy stream listener which is defined in the custom_stream_listener module.
- **live_get_streamer()** accepts a tweepy stream listener and a track (a list of words which Twitter uses to know what to send back to a user). The return value is an active tweepy stream.
- **disk_get_tweet_queue()** returns a pickled list of tweepy tweets.

generate_wordcloud(module)

Path: root/packages/graphical/generate_wordcloud
Description: This is a collection of functions which are used for generating wordclouds.

Functions:

- **get_long_tweet_objects()** takes a path string as an argument and uses it to return a list of cleaned dataobjects.
- **get_long_tweet_string()** takes a path string as an argument, uses get_long_tweet_objects() to get a list of cleaned dataobjects, and returns a string made from the text field of those dataobjects.
- **generate_wordcloud()** takes a path string as an argument and uses get_long_tweet_string() to create a simple wordcloud.
- **write_csv()** accepts an output path string and a list of dataobjects which are written as a CSV to the specified path.

base(module)

Path: root/packages/pipes/collection/base
Description: Contains base class for pipes.

Methods:

- **init()** has the following parameters:
  - previous_pipe; a subclass of self.
  - process_task; a function which is wrapped in self.process()
  - threshold_output; signals the max size of self.output
  - verbosity; used for conditional printout.

  This method sets the arguments as properties and creates a self.output.
- **cond_print()** takes in a string as an argument and prints it out if self.verbosity is True.
- **clear_overflow()** removes objects from self.output if the object count in that list is more than self.threshold_output.
- **process()** attempts to get data from self.previous_pipe, transform it with self.process_task and move it to self.output.

cleaning(module)

Path: root/packages/pipes/collection/cleaning
Description: Contains a pipe class which converts tweepy tweets into cleaned dataobjects.

Methods:

- **init()** accepts all parameters needed for PipeBase class (see 'base' module), except for process_task, which is pointed to the method self.task.
- **task()** is intended to be called from process() in base class. It accepts an element which is expected to be a tweepy tweet, converts it to a dataobject, cleans it, and returns it.

database(module)

Path: root/packages/pipes/collection/database

Description: Contains a pipe which is a top-level interface for all database tools. Purpose of this pipe is to insert dataobjects from previous_pipe_output into the database and do sorting along the way.

Methods:

- **init()** accepts all parameters needed for PipeBase class (see 'base' module), except for process_task, which is pointed to the method self.task. Additionally, a 'start_fresh' bool used (param) to signal whether or not the database should be cleared before use.
- **setup()** gets all necessary database tools and determines whether or not a root ring exists in the database. If that is not the case, then a new one will be created eventually.
- **task()** is intended to be called from process() in base class. It accepts an element which is expected to be a dataobject with a siminet. Database insertion and sorting is done here.

feed_api(module)

Path: root/packages/pipes/collection/feed_api

Description: Contains a pipe which is used to set up and start a stream from Twitter API.

Methods:

- **init()** accepts all parameters needed for PipeBase class (see 'base' module), except for process_task, which is pointed to the method self.task. 'previous_pipe' isn't accepted either, as this pipe is intended to be first in a pipeline. Additionally, a 'track' parameter is accepted for telling the API what to send back.
- **set_stream_fetch_input()** sets up and starts the steam.
- **task()** no function.
- **del()** shuts down the tweepy stream on garbage collection.

feed_disk(module)

Path: root/packages/pipes/collection/feed_api

Description: Contains a pipe which is used to stream tweepy tweets from a dataset generated with 'generate_dataset'(module).

Methods:

- **init()** accepts all parameters needed for PipeBase class (see 'base' module), except for process_task, which is pointed at the method self.task. 'previous_pipe' isn't accepted either, as this pipe is intended to be first in a pipeline. Additionally, a 'filepath' is accepted to specify where a dataset is stored.
- **get_unpickle_generator()** gives back a generator which is used to stream tweepy tweets from the path specified as a parameter.
- **task()** is intended to be called from process() in base class and accepts an object(any, but not used). Each call uses the generator created by get_unpickle_generator and set in init to return objects.

pyjs_bridge(module)

Path: root/packages/pipes/collection/pyjs_bridge

Description: Contains a pipe which accepts and converts a query into a siminet, then compares that to the siminet of incoming dataobjects. The result of that comparison is then sent over Websockets to a front-end.

Methods:

- **init()** accepts all parameters needed for PipeBase class (see 'base' module), except for process_task, which is pointed at the method self.task. Additionally, 'query' is accepted and expected to be a list of strings.
- **set_simitool()** attempts to find a simitool (creates siminets) in the previous_pipe (property) stack. On fail, a new simitool is instantiated.
- **start()** starts a webserver using asyncio and websockets, using self.serve_loop() as a callback/eventloop.
- **serve_loop()** is used by websockets as a callback. This method is also used as an eventloop where calls to other methods are done (check_query_update and calc_next_score).
- **confirm_str_lst()** accepts a list and checks if all of the items in that list contains strings.
- **check_query_update()** checks if a new query is set to self.query_queued. If so, that query will be processed and moved to self.query_ready.
- **calc_next_score()** compares the siminet of self.query_ready against the siminet of dataobjects in self.foreign_data_queue (if any). Returns a similarity score.
- **task()** is intended to be called from process() in the base class and accepts a type which is expected to be a dataobject with a siminet. The first time this method is called, a new thread is spawned to call self.start(). This method also moves data given as an argument to self.foreign_data_queue.

simi(module)

Path: root/packages/pipes/collection/simi

Description: A pipe used to attach simi nets to dataobjects

Methods:

- **init()** accepts all parameters needed for PipeBase class (see 'base' module), except for process_task, which is pointed at the method self.task. Additionally, a 'recursion_level' parameter is used to specify how many levels a siminet should have. Setup of simi tool is done here.
- **task()** is intended to be called from process() in base class and accepts a type which is expected to be a dataobject. That dataobject gets a siminet before being returned.

pipeline(module)

Path: root/packages/pipes/pipeline

Description: This module contains a class which is used to collect and run pipes.

Methods:

- **init()** accepts a lists of pipes.
- **monitor_pipes()** does a printout of pipes collected in this class with the object count of their output lists. Text is usually replaced (a bug exists at the time of writing where this is not the case. caused by using pyjs_bridge pipe).
- **run()** calls process() of all pipes collected in this class in an infinite loop. self.monitor_pipes() is also called from here.

prefabs(module)

Path: root/packages/pipes/prefabs
Description: Collection of pipeline prefabs.

Functions:

- **get_pipeline_api_cln_simi_db()** accepts the following arguments:
  - api_track; tells Twitter API what to send.
  - rec_lvl; recursion level for siminets.
  - threshold_output; max obj count in a pipe.
  - verbosity; printout in pipes.

  Creates and returns a pipeline instance containing pipes in these modules: feed_api, cleaning, simi and database.
- **get_pipeline_dsk_cln_simi_db()** accepts the following arguments:
  - filepath; points to a dataset created with the module generate_dataset.
  - rec_lvl; recursion level for siminets.
  - threshold_output; max obj count in a pipe.
  - verbosity; printout in pipes.

  Creates and returns a pipeline instance containing the pipes in these modules: feed_disk, cleaning, simi and database.
- **get_pipeline_dsk_cln_simi_js()** accepts the following arguments:
  - filepath; points to a dataset created with the module generate_dataset.
  - initial_query; what the pyjs_bridge will use for comparisons against incoming dataobjects.
  - rec_lvl; recursion level for siminets.
  - threshold_output; max obj count in a pipe.
  - verbosity; printout in pipes.

  Creates and returns a pipeline instance containing the pipes in these modules: feed_disk, cleaning, simi and pyjs_bridge.
- **get_pipeline_api_cln_simi_js()** accepts the following arguments:
  - api_track; tells Twitter API what to send.
  - initial_query; what the pyjs_bridge will use for comparisons against incoming dataobjects.
  - rec_lvl; recursion level for siminets.
  - threshold_output; max obj count in a pipe.
  - verbosity; printout in pipes.

  Creates and returns a pipeline instance containing the pipes in these modules: feed_api, cleaning, simi and pyjs_bridge

process_tools(module)

Path: root/similarity/process_tools
Description: Is used to create siminets and do actions related to similarity comparisons.

Methods:

- **init()** only takes in a verbosity argument, used for conditional printing calls in other methods.
- **get_model_info()** takes in a string to get info about some gensim model.
- **cond_print()** takes in a string parameter which is printed if self.verbosity is True.
- **load_model()** takes in a string as an argument. Uses it to load a model to self.
- **get_similarity_net()** takes in the following arguments:
  - query; a list of strings which can be used to create a single similarity net.

    - max_recursion; expected to be an int. Determines the recursion level of the similarity net builder. Each word in the query list will yield 10 similar words for each recursion.

  This method is a wrapper for a recursive similarity net builder. Similarity net compression is enforced on the return line.
- **compress_similarity_net()** expects a similarity net as an argument, which is compressed such that duplicate words are removed (without affecting the overall cumulative confidence score). Result is returned.
- **get_score_compressed_siminet()** accepts two compressed similarity nets and returns a similarity score.
- **get_top_simi_index()** accepts two arguments;
    - new_object; a dataobject which is expected to have a similarity net.
    - other_objects; a list of dataobjects which are expected to have similarity nets.

  The similarity net of all other_objects will be calculated against the similarity net of new_object. An index can be returned, pointing to the object in other_objects which has the highest similarity comparison to new_object. If nothing is similar, None will be returned.
- **get_representative()** expects a list of dataobjects with siminets attached. This method will sort the list by similarity, meaning that the object on index 0 should be most similar to all other objects. Returns a list where index 0 indicates sorting status, and index 1 is the new sorted list. Sorting status can be False if this sorting method didn't change the order of the input list (naturally in order?).