Bachelor Thesis Documentation

Release v1.0

Bachelor Thesis

Jul 28, 2016

CONTENTS

Contents:

CHAPTER

ONE

BACHELORARBEIT

1.1 src package

1.1.1 Subpackages

src.clustering package

Submodules

src.clustering.cluster_mappings module

Script to cluster mapping vectors created with *src.mapping.mapthreading* .

src.clustering.cluster_mappings. **aggregate_cluster** (*points*, *labels*)

Arranges all clusters in a list, where a sublist with all points at index i corresponds with the custer with label i.

Parameters

- **points** (*list*) – List of datapoints
- **labels** (*list*) – List of unique cluster labels

Returns list of lists of datapoints belonging to the i-th cluster

Return type list

src.clustering.cluster_mappings. **cluster_mappings** (*vector_inpath*, *do_pca=False*, *target_dim=100*, *indices_inpath=None*, *epsilon=2.625*, *min_s=20*)

Cluster mapping vectors created with *src.mapping.mapthreading* or rc.mapping.map_vectors.py . Because just reading about the number of clusters and their sizes, there's an option to resolve the indices of the vectors in the cluster to their original word pairs.

Parameters

- **vector_inpath** (*str*) – Path to vector file. File should have the following format (separated by spaces): <index of original vector #1> <index of original vector #2> <Dimension 1> ... <Dimension n>
- **do_pca** (*bool*) – Flag to indicate whether PCA should be executed before clustering to reduce amount of
- **computation.** –

- **target_dim** (*int*) – Number of dimensions vectors should be shrunk to in case PCA is performed.
- **indices_inpath** (*str*) – Path to file with the indices given to words. The file should have the following format: <index of word> <word> (separated by tab)
- **epsilon** (*float*) – Radius of circle DBSCAN uses to look for other data points.
- **min_s** (*int*) – Minimum number of points in radius epsilon DBSCAN needs to declare a point a core object.

src.clustering.cluster_mappings. **get_cluster_size** (*labels*)
Calculate the size of every cluster found by DBSCAN.

Parameters **labels** (*list*) – List of cluster IDs assigned to every data point.

Returns Dictionary of cluster sizes with cluster id as key and cluster size as value.

Return type defaultdict

src.clustering.cluster_mappings. **init_argparser** ( )
Initialize all possible arguments for the argument parser.

Returns ArgumentParser object with command line arguments for this script.

Return type argparse.ArgumentParser

src.clustering.cluster_mappings. **load_indices** (*indices_inpath*)
Load word indices from a file. The file should have the following format: <index of word> <word> (separated by tab)

Parameters **indices_inpath** (*str*) – Path to index file.

src.clustering.cluster_mappings. **load_mappings_from_model** (*mapping_inpath*)
Load mapping vectors from file.

Parameters **mapping_inpath** – Path mapping vector file.

Returns A tuple of a list of word index pairs and a dictionary (defaultdict) with index pair tuple as key and mapping vector (as numpy.array) as value.

Return type tuple

src.clustering.cluster_mappings. **main** ( )
This is the main function. It uses the parsed command line arguments to pick the right function to execute.

src.clustering.cluster_mappings. **resolve_indices** (*points*, *labels*, *indices_inpath*, *model*)

src.clustering.cluster_mappings. **train_clustering_parameters** (*vector_inpath*)
Functions that tries to figure out the optimal clustering parameters in regard to DBSCAN's epsilon, min_samples and p.

Parameters **vector_inpath** (*str*) – Path to vector file. File has to have the following format (separated by spaces): <index of original vector #1> <index of original vector #2> <Dimension 1> ... <Dimension n>

Module contents

src.eval package

Submodules

src.eval.analogy module

Module to evaluate word embeddings by the means of analogies like "W is to X like Y is to Z". Usually, the system uses the word embeddings of word W, X, Y and tries to find the vector of word Z that is most similar to X and Y and most dissimilar to W. Therefore, the CosMul method (Levy et al., 2015) is used.

The whole module is used in `src.eval.eval_vectors.py`.

`src.eval.analogy.` **`analogy_eval`** (*vector_inpath*, *analogy_path*, *per_section=False*)

Perform analogy evaluation. Usually, the system uses the word embeddings of word W, X, Y and tries to find the vector of word Z that is most similar to X and Y and most dissimilar to W for an analogy like "W is to X like Y is to Z." Therefore, the CosMul method (Levy et al., 2015) is used.

Parameters

- **`vector_inpath`** (*str*) – Path to *word2vec* vector file.
- **`analogy_path`** (*str*) – Path to analogy file.
- **`per_section`** (*bool*) – Flag to indicate whether analogies test should be conducted section-wise or just all in one run.

`src.eval.analogy.` **`read_analogies`** (*analogy_path*, *per_section=False*)

Reads a file with analogies.

Parameters

- **`analogy_path`** (*str*) – Path to analogy file.
- **`per_section`** (*bool*) – Flag to indicate whether analogies test should be conducted section-wise or just all in one run. In this function, the section will be put into a data structure accordingly.

Returns Dictionary with section header as key, list of analogy as 4-tuples as value.

Return type dict

src.eval.eval_vectors module

Main module used to evaluate word embeddings. It offers the following options:

1.) Analogy: The system tries to complete an analogy like "W is to X like Y is to...?" The percentage of correct answers is measured.

2.) Word similarity: The system assign word pairs a similarity score based on the cosine similarity of their word embeddings. Then, to correlation between those and human ratings is measured with Pearson's rho.

3.) Nearest neighbors: Find the nearest neighbors for a list of words based on their word embeddings. Good for a first look on the data, but not quantifiable.

4.) Visualize: Plot word embeddings in 2D or 3D. Fancy plots. Yay!

`src.eval.eval_vectors.` **`find_nearest_neighbors`** (*vector_inpath*, *max_n*, *wordlist*)

Find the nearest neighbors for a list of words based on their word embeddings.

Parameters

- **vector_inpath** (*str*) – Path to vector file. File has to have the following format (separated by spaces): <index of original vector #1> <index of original vector #2> <Dimension 1> ... <Dimension n>
- **max_n** (*int*) – Number of nearest neighbors that should be determined.
- **wordlist** (*list*) – List of words nearest neighbors should be found for.

src.eval.eval_vectors. **init_argparser** ( )
Initialize all possible arguments for the argument parser.

Returns ArgumentParser object with command line arguments for this script.

Return type argparse.ArgumentParser

src.eval.eval_vectors. **main** ( )
This is the main function. It uses the parsed command line arguments, especially *–mode*, to pick the right function to execute.

src.eval.eval_vectors. **plot** (*vector_inpath*, *max_n*, *target_dim*, *show_plot=False*, *display_names=False*)
Plot word embeddings in 2D or 3D. As a heuristic, word will only be plotted after the 50th most frequent words to avoid plotting boring stop words.

Parameters

- **vector_inpath** (*str*) – Path to vector file. File has to have the following format (separated by spaces): <index of original vector #1> <index of original vector #2> <Dimension 1> ... <Dimension n>
- **max_n** (*int*) – Maximum number of vectors to be plotted.
- **show_plot** (*bool*) – Flag to indicate whether a window with the (interactive) plot should pop up after executing the script.
- **display_names** (*bool*) – Flag to indicate whether the words should acutally be shown next to the data point in the plot. Can get very messy with higher *max_n*.

src.eval.word_similarity module

Module used to conduct the word similarity evaluation. The system assign word pairs a similarity score based on the cosine similarity of their word embeddings. Then, to correlation between those and human ratings is measured with Pearson's rho.

The whole module is used in src.eval.eval_vectors.py .

src.eval.word_similarity. **evaluate_wordpair_sims** (*x*, *y*, *number_of_pairs*)
Evaluate results of the similarity score assignments, i.e. calculate pearson's rho and its significance.

Parameters

- **x** (*list*) – List of similarity scores assigned by humans.
- **y** (*list*) – List of similarity scores assigned by the system.
- **number_of_pairs** (*int*) – Number of word pairs evaluated.

Returns **rho** – Pearson's correlation coefficient. t (float): Student's t value. z (float): z value.

Return type float

src.eval.word_similarity. **read_wordpairs** (*wordpair_path*, *format='google'*)
Read wordpair file with wordpairs and their similarity scores assigned by humans.

Parameters

- **wordpair_path** (*str*) – Path to word pair file.
- **format** (*str*) – Format of wor pair file {google|semrel}

Returns Tuple of a list of word pairs and a list of similarity scores for those same pair assigned by humans.

Return type tuple

src.eval.word_similarity. **remove_unknowns** (*x*, *y*)

Remove word pairs from the results where one or two word embedding weren't found.

Parameters

- **x** (*list*) – List of similarity scores assigned by humans.
- **y** (*list*) – List of similarity scores assigned by the system.

Returns x – Purged list of similarity scores assigned by humans. y (list): Purged list of similarity scores assigned by the system.

Return type list

src.eval.word_similarity. **word_sim_eval** (*vector_inpath*, *wordpair_path*, *format='google'*)

Function that let's the system assign word pairs a similarity score based on the cosine similarity of their word embeddings. Then, to correlation between those and human ratings is measured with Pearson's rho.

Parameters

- **vector_inpath** (*str*) – Path to vector file. File has to have the following format (separated by spaces): <index of original vector #1> <index of original vector #2> <Dimension 1> ... <Dimension n>
- **wordpair_path** (*str*) – Path to word pair file.
- **format** (*str*) – Format of word pair file {google|semrel}

Module contents

src.mapping package

Submodules

src.mapping.mapthreading module

Module used to map a pair of vectors into a new combined vector space. Those mappings will be created by multiple threads in a master-slave-pattern. To do so, the user can choose between different vector operations as offset, cosine similarity, euclidean distance and many more.

Warning: Because of $\Omega = \frac{n(n-1)}{2}$, it is recommended to use the co-occurrence constraint Λ, which limits the calculations to word embedding pairs which words occurred together in a corpus in at least *n* sentences (but it will still take quite a while).

class src.mapping.mapthreading. **MappingMasterThread** (*n*, *vector_inpath*, *vector_outpath*, *features*, *lambda_*, *ids_inpath*, *indices_inpath*)

Bases: threading.Thread

Master thread class. The master thread loads all necessary data into suitable data structures and distributes them among all worker threads.

prepare ( )

Loads The master thread loads all necessary data into suitable data structures. To be more specific, word embeddings, sentence IDs and word indices are processed.

read_ids_file (*ids_inpath*)

Read the sentence ID file.

Parameters **ids_inpath** (*str*) – Path to sentence IDs file. The file should be in the following *YAML*-format: - <word>:

- <sentence id>
- <sentence id>

...

Returns Dictionary with words as keys and the IDs of the sentences they occur in in a set as value.

Return type defaultdict

start_threads ( )

Starts all the threads (and ends them if they're all finished).

class src.mapping.mapthreading. **MappingWorkerThread** (*worker_id*, *vector_dict*, *vector_queue*, *vector_outpath*, *features*, *occurrences*, *indices*, *lambda_*)

Bases: threading.Thread

Worker thread class. The worker threads do all the dirty work after they receive all necessary data from the master thread an try to calculate every possible combinations of two word embeddings in a dataset.

All the word embeddings will be stores in a dictionary (*VectorDict* as well as a Queue). An idle thread picks a new vector from the queue and then starts to iterate over all the vectors in the *VectorDict* (this way, the queue gets shorter over time while the size of the dictionary stays fixed).

Before it starts calculations, it checks a) if the co-occurrence constraint is satisfied and b) if this combination of word embeddings has already been processed.

cosine_similarity (*v1*, *v2*)

Calculates the cosine similarity ($cos(\vec{v}_1, \vec{v}_2) \in [-1, -1]$) between two vectors.

Parameters

- **v1** (*numpy.array*) – First vector
- **v2** (*numpy.array*) – Second vector

Returns Cosine similarity between the two vectors.

Return type float

distance (*v1*, *v2*)

Return the vector offset of two vectors:

Parameters

- **v1** (*numpy.array*) – First vector
- **v2** (*numpy.array*) – Second vector

Returns numpy.array: Vector offset.

euclidean_distance1 (*v1*, *v2*)

Return the euclidean distance between two vectors.

$$eucl(\vec{a}, \vec{b}) = \sqrt{\sum_{i=1}^{n} (\vec{b}_i - \vec{a}_i)^2}$$

Parameters

- **v1** (*numpy.array*) – First vector
- **v2** (*numpy.array*) – Second vector

Returns Euclidean distance between the two vectors.

Return type float

euclidean_distance2 (*v1*, *v2*)

Returns the squared euclidean distance between two vectors.

$$eucl2(\vec{a}, \vec{b}) = \sum_{i=1}^{n} (\vec{b}_i - \vec{a}_i)^2$$

Parameters

- **v1** (*numpy.array*) – First vector
- **v2** (*numpy.array*) – Second vector

Returns Squared euclidean distance between the two vectors.

Return type float

hash_indices (*i1*, *i2*)

Combines two vector indices (the indices of the words' embeddings used in vector operations) into a hash s.t. threads can do an easy lookup if a mapping vector has already been calculated. To guarantee this, $h(i_1, i_2) = h(i_2, i_1)$ has to be the case.

Parameters

- **i1** (*int*) – Index of first word's embedding
- **i2** (*int*) – Index of second word's embedding

Returns Unique hash for index pair.

Return type int

manhattan_distance (*v1*, *v2*)

Returns the manhattan distance between two vectors.

$$manhattan(\vec{a}, \vec{b}) = \sum_{i=1}^{n} |\vec{b}_i - \vec{a}_i|$$

Parameters

- **v1** (*numpy.array*) – First vector
- **v2** (*numpy.array*) – Second vector

Returns Manhattan distance between the two vectors.

Return type float

run ( )
Starts a worker thread.

soft_cosine_similarity (*v1*, *v2*)
Calculates the soft cosine similarity between two vectors.

$$S = \begin{bmatrix} eucl(\vec{a}_1, \vec{b}_1) & \dots & eucl(\vec{a}_1, \vec{b}_n) \\ \vdots & \ddots & \vdots \\ eucl(\vec{a}_n, \vec{b}_1) & \dots & eucl(\vec{a}_n, \vec{b}_n) \end{bmatrix}$$

$$softcos(\vec{a}, \vec{b}) = \frac{\sum_{i,j}^{N} S_{ij} \vec{a}_i \vec{b}_j}{\sqrt{\sum_{i,j}^{N} S_{ij} \vec{a}_i \vec{a}_j} \sqrt{\sum_{i,j}^{N} S_{ij} \vec{b}_i \vec{b}_j}}$$

(It considers the similarity between pairs of features.)

Parameters

- **v1** (*numpy.array*) – First vector
- **v2** (*numpy.array*) – Second vector

Returns Soft cosine similarity between the two vectors.

Return type float

class src.mapping.mapthreading. **VectorDict**
Bases: object

VectorDict class that serves two functions:

1.) Storing word embeddings so they don't allocate memory for every worker thread
2.) Providing a set, where are processed vector pairs are stored so no redundant computations are made.

Locks are used for synchronization purposes.

add_skippable (*index_hash*)
Add the hash of an index pair to a set of already processed vector pairs.

Parameters **index_hash** (*int*) – Hash value of index pair. Produced with hash_indices() .

add_vector (*index*, *vector*)
Add a new word embedding.

Parameters

- **index** (*int*) – Index of the word the embedding belongs to.
- **vector** (*numpy.array*) – Word embedding corresponding to given index.

get_keys ( )
Get all the keys (word embedding IDs) of this dictionary.

Returns List of word embedding IDs.

Return type list

get_vector (*index*)
Get a word embedding given its word's index.

Parameters **index** (*int*) – Index of the word the embedding belongs to.

Returns Word embedding corresponding to given index.

Return type numpy.array

skippable (*index_hash*)
Checks whether a pair of vectors has already been processed.

Parameters **index_hash** (*int*) – Hash value of index pair. Produced with `hash_indices()`.

Returns Whether a pair of vectors has already been processed.

Return type bool

src.mapping.mapthreading. **alt** (*func*)
Prepends the local time to the output of a function.

Parameters **func** (*function*) – Function the local time should be prepended to.

src.mapping.mapthreading. **init_argparse** ( )
Initialize all possible arguments for the argument parser.

Returns ArgumentParser object with command line arguments for this script.

Return type `argparse.ArgumentParser`

src.mapping.mapthreading. **main** ( )
Main function that initializes the master thread with command line arguments and starts it.

Module contents

src.misc package

Submodules

src.misc.decorators module

src.misc.decorators. **alt** (*func*)

src.misc.decorators. **log_time** (*logpath='log.txt'*, *interval=5*)

src.misc.decorators. **log_time_mp** (*logpath='log.txt'*, *interval=5*)

src.misc.helpers module

src.misc.helpers. **alt** (*func*)
Prepends the local time to the output of a function.

Parameters **func** (*function*) – Function the local time should be prepended to.

src.misc.helpers. **capitalize** (*word*)

src.misc.helpers. **load_vectors_from_model** (*vector_inpath*, *max_n=None*, *logpath=None*, *indices=False*)

Module contents

src.prep package

Subpackages

src.prep.corpus package

Submodules

src.prep.corpus.convert_to_plain module

Convert the *DECOW14X* corpus into a plain text file. Is used as pre-processing step for the word2vec training. To make this this more feasible (decow is a **huge** corpus), python's `multiprocessing` is used, s.t. every part of the corpus in simultaneously processed. Afterwards, a bash command like `cat` can be used to merge into one single file.

`src.prep.corpus.convert_to_plain.` **`contains_tag`** (*line*)
Checks whether the current line contains an xml tag.

Parameters **`line`** (*str*) – Current line

Returns Whether the current line contains an xml tag.

Return type bool

`src.prep.corpus.convert_to_plain.` **`convert_decow_to_plain`** (*decow_dir*, *out_dir*, *log_path*, *merge_nes*, *log_interval*)
Convert the whole corpus into plain text.

Parameters

- **`decow_dir`** (*str*) – Path to directory with decow corpus paths.
- **`out_dir`** (*str*) – Path where plain text parts should be written to.
- **`log_path`** (*str*) – Path where the log files should be written to.
- **`merge_nes`** (*bool*) – Flag to indicate whether multi-word expression should be merged with underscores.
- **`log_interval`** (*int*) – Interval to log current process state in seconds.

`src.prep.corpus.convert_to_plain.` **`convert_part`** (*argstuple*)
Convert a corpus part into plain text without merging multiple word entries.

Parameters **`argstuple`** – Tuple of methods arguments (`inpath` (*str*): Path to this processes' corpus part / `dir_outpath` (*str*): Path to this processes' output / `log_path` (*str*): Path to this processes' log / `interval` (*int*): Logging interval in seconds)

`src.prep.corpus.convert_to_plain.` **`convert_part_merging`** (*argstuple*)
Convert a corpus part into plain text and merging multiple word entries.

Parameters **`argstuple`** – Tuple of methods arguments (`inpath` (*str*): Path to this processes' corpus part / `dir_outpath` (*str*): Path to this processes' output / `log_path` (*str*): Path to this processes' log / `interval` (*int*): Logging interval in seconds)

`src.prep.corpus.convert_to_plain.` **`extract_named_entity`** (*line*)
Extract named entity from current line.

Parameters **line** (*str*) – Current line

Returns Extracted named entity or None if no named entity is present.

Return type str or None

src.prep.corpus.convert_to_plain. **extract_sentence_id** (*tag*)
Extract the sentence ID of current sentence.

Parameters **tag** (*str*) – Sentence tag

Returns sentence ID

Return type str

src.prep.corpus.convert_to_plain. **get_file_number** (*filename*)
Get the number of the current decow corpus part.

Parameters **filename** (*str*) – Decow corpus part file name

Returns File number

Return type str

src.prep.corpus.convert_to_plain. **main** ( )
Main function. Uses command lines to start corpus processing.

src.prep.corpus.extract_conll module

This script can be used to extract information out of a specific column of a file in the CoNLL-format.

src.prep.corpus.extract_conll. **extract_conll** (*inpath*, *outpath*, *column*)
Extract information out of CoNLL files.

Parameters

- **inpath** (*str*) – Path to input file.
- **outpath** (*str*) – Path to output file.
- **column** (*int*) – The number (-1) of the column the information should be extracted from.

src.prep.corpus.extract_conll. **init_argparse** ( )
Initialize all possible arguments for the argument parser.

Returns ArgumentParser object with command line arguments for this script.

Return type `argparse.ArgumentParser`

src.prep.corpus.extract_conll. **main** ( )
The main function.

src.prep.corpus.mapper module

Mapper classed used to count frequencies of words in a corpus. Corpus has to be in plain text format. This class is used in a Map-Reduce-pattern, so you also need the `reducer.py` class.

Then, you can open your terminal and pipe them together:

```
> cat corpus.txt | ./mapper.py | sort | ./reducer.py
```

Also, you probably have to remove the `if __name__ == "__main__":` line and unindent the remaining code, this is only due to sphinx being picky and not documenting plain python scripts at all.

src.prep.corpus.reducer module

Reducer classed used to count frequencies of words in a corpus. Corpus has to be in plain text format. This class is used in a Map-Reduce-pattern, so you also need the `mapper.py` class.

Then, you can open your terminal and pipe them together:

```
> cat corpus.txt | ./mapper.py | sort | ./reducer.py
```

Also, you probably have to remove the `if __name__ == "__main__":` line and unindent the remaining code, this is only due to sphinx being picky and not documenting plain python scripts at all.

Module contents

src.prep.nes package

Submodules

src.prep.nes.extractNE module

src.prep.nes.extractNE. **contains_tag** (*line*)

src.prep.nes.extractNE. **extract_named_entity** (*line*)

src.prep.nes.extractNE. **extract_sentence_id** (*tag*)

src.prep.nes.extractNE. **main** ( )

src.prep.nes.extractNE. **print_dict_in_file** (*dictionary*, *out_path*)

src.prep.nes.extractNE. **print_ids_in_file** (*dictionary*, *out_path*)

src.prep.nes.extractNE. **print_list_in_file** (*ne_list*, *out_path*)

src.prep.nes.extractNE. **process** (*inpath*, *outpath*, *logpath*)

src.prep.nes.merge module

src.prep.nes.merge. **dump_ids_dict** (*idsdict*, *outpath*)

src.prep.nes.merge. **freqWorker** (*inpath*)

src.prep.nes.merge. **idWorker** (*inpath*)

src.prep.nes.merge. **load_ids_dict** (*inpath*)

src.prep.nes.merge. **main** ( )

src.prep.nes.merge. **mergeDicts** (*dicttuple*)

src.prep.nes.merge. **merge_frequency_files** (*infiles_path*, *outpath*, *logpath*)

src.prep.nes.merge. **merge_id_dicts** (*dicttuple*)

src.prep.nes.merge. **merge_id_files** (*infiles_path*, *outpath*, *logpath*, *yaml=False*)

src.prep.nes.merge. **print_key_lengths** (*dictionary*)

src.prep.nes.merge. **rl** (*infile*)

src.prep.nes.mwe module

src.prep.nes.mwe. **create_mwe_pickle** (*inpath*, *outpath*, *logpath='./mwes.log'*)

src.prep.nes.mwe. **create_mwe_pickle2** (*inpath*, *outpath*, *logpath='./mwes.log'*)

src.prep.nes.mwe. **dump_dict_pickle** (*d*, *outpath*)

src.prep.nes.mwe. **dump_dict_pickle2** (*d*, *outpath*)

src.prep.nes.mwe. **load_dict_pickle** (*inpath*)

src.prep.nes.mwe. **load_dict_pickle2** (*inpath*)

src.prep.nes.mwe. **main** ( )

src.prep.nes.mwe. **replace_mwes** (*mwe_path*, *corpus_path*, *out_path*)

src.prep.nes.statistics module

src.prep.nes.statistics. **calculate_occurrences** (*freqpath*, *relations_path*)

src.prep.nes.statistics. **main** ( )

Module contents

src.prep.relations package

Submodules

src.prep.relations.relations module

exception src.prep.relations.relations. **MissingTranslationException**

Bases: exceptions.Exception

get_id ( )

src.prep.relations.relations. **fetch_name** (*id*, *lang='en'*)

src.prep.relations.relations. **fetch_relation_triples_of_file** (*inpath*, *outpath*, *logpath*, *lang='en'*)

src.prep.relations.relations. **format_fbid** (*id*)

src.prep.relations.relations. **freebase_request** (*query*, *api_key*, *service_url*)

src.prep.relations.relations. **main** ( )

src.prep.relations.relations. **read_credentials** ( )

src.prep.relations.relations. **rl** (*infile*)

src.prep.relations.relations. **translate_name** (*name*, *lang='en'*)

src.prep.relations.relations. **translate_word2vec_question_phrases** (*inpath*, *outpath*, *lang='en'*)

Module contents

Module contents

src.trans_e package

Submodules

src.trans_e.add_inverse_relations module

src.trans_e.add_inverse_relations. **add_inverse_relations** (*relations_inpath*, *relations_outpath*, *inverse_relations*, *known_relations*)

src.trans_e.add_inverse_relations. **init_argparse** ( )

src.trans_e.add_inverse_relations. **main** ( )

src.trans_e.add_inverse_relations. **read_file_with_inverse_relations** (*inverse_inpath*)

src.trans_e.clean_relations module

src.trans_e.contains_entities module

src.trans_e.contains_entities. **contains_entities** (*entities1*, *entities2*)

src.trans_e.contains_entities. **create_new_dataset** (*entities1*, *dataset*, *outpath*)

src.trans_e.contains_entities. **extract_entities_from_relation_dataset** (*dataset_inpath*)

src.trans_e.contains_entities. **extract_entities_from_tql_file** (*tql_path*)

src.trans_e.contains_entities. **format_fbid** (*id*)

src.trans_e.contains_entities. **init_argparse** ( )

src.trans_e.contains_entities. **main** ( )

src.trans_e.convert_relations module

src.trans_e.differentiate_datasets module

src.trans_e.differentiate_datasets. **compare_entities** (*set1*, *set2*)

src.trans_e.differentiate_datasets. **init_argparse** ( )

src.trans_e.differentiate_datasets. **main** ( )

src.trans_e.differentiate_datasets. **read_dataset** (*inpath*)

src.trans_e.partition_data module

src.trans_e.partition_data. **check_data_integrity** (*data_inpath*, *remove_clones*, *outpath*)

Check whether all triplets in the data are unique.

src.trans_e.partition_data. **check_set_integrity** (*indir*)

src.trans_e.partition_data. **get_stats** (*data*)

src.trans_e.partition_data. **init_argparse** ( )

src.trans_e.partition_data. **main** ( )

src.trans_e.partition_data. **partition_data** (*data*, *prts*, *outdir*, *whole=True*)

src.trans_e.partition_data. **partition_relation_wise** (*data*, *prts*)

src.trans_e.partition_data. **partition_whole** (*data*, *prts*)

src.trans_e.partition_data. **partitions_list** (*l*, *prts*)

src.trans_e.partition_data. **read_only_relations_into_set** (*inpath*)

src.trans_e.partition_data. **read_relations** (*inpath*)

src.trans_e.partition_data. **write_data_in_file** (*data*, *outfile*)

src.trans_e.trans_we module

src.trans_e.trans_we. **convert_data** (*sets_path*, *tql_inpath*, *vector_inpath*)

src.trans_e.trans_we. **create_corrupt_triples** (*grouped_pairs*, *entities*)

src.trans_e.trans_we. **dump_relation_vectors** (*relation_vectors*, *outpath*)

src.trans_e.trans_we. **evaluate** (*model*, *grouped_test*, *relation_vectors*, *entities*)

src.trans_e.trans_we. **extract_data_from_uri** (*uri*)

src.trans_e.trans_we. **get_rank** (*target*, *ranks*)

src.trans_e.trans_we. **init_argparser** ( )

Initialize all arguments for an ArgumentParser object and return it.

@returns {ArgumentParser} argument parser object

src.trans_e.trans_we. **load_relation_vectors** (*inpath*)

src.trans_e.trans_we. **load_vectors** (*vector_inpath*)

@param vector_inpath: Path to word2vec model file

src.trans_e.trans_we. **main** ( )

src.trans_e.trans_we. **prepare_training** (*sets_path*, *vector_inpath*)

src.trans_e.trans_we. **rank_entities** (*reference*, *solution*, *model*, *entities*)

src.trans_e.trans_we. **read_freebase_data** (*sets_path*)

src.trans_e.trans_we. **read_freebase_file** (*fb_inpath*)

src.trans_e.trans_we. **read_tql_file** (*tql_inpath*)

src.trans_e.trans_we. **test_coverage** (*triples*, *model*)
Test the coverage of a dataset consisting of freebase triples on word2vec word embeddings. For every triple (h, l, t), the entities h and t are taken and used for look up in the word2vec model.

@param triples: list of 3-tuples (freebase triples) @param model: gensim word2vec model

src.trans_e.trans_we. **train** (*model*, *grouped_train*, *grouped_corrupted*, *lossf*, *relation_types*, *epochs=1000*, *learning_rate=0.01*, *margin=1*)

src.trans_e.trans_we. **transform_triples** (*triples*, *relation_types*, *entities*)

src.trans_e.trans_we. **write_data** (*triples*, *found_entities*, *outpath*)

Module contents

1.1.2 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

S

A

C

D

E

F

G

H

I

L

M

P

R

S

T

V

W