Agatha

Release 2020-04-29

Justin Sybrandt, Ilya Tyagin

Jun 03, 2020

CONTENTS

CHAPTER

ONE

AGATHA OVERVIEW

Checkout our Docs on ReadTheDocs

We are currently doing a bunch of development around the CORD-19 datsset. These customizations have been funded by an NSF RAPID grant. *Follow Along with Development on trello*.

If you're here looking for the *CBAG: Conditional Biomedical Abstract Generation* project, take a look in the agatha/ml/abstract_generator submodule.

1.1 Install Agatha to use pretrained models

In our paper we present state-of-the-art performance numbers across a range of recent biomedical discoveries across popular biomedical sub-domains. We trained the Agatha system using only data published prior to 2015, and supply the necessary subset of that data in an easy-to-replicate package. Note, the full release is also available for those wishing to tinker further. Here's how to get started.

Setup a conda environment

```
conda create -n agatha python=3.8
conda activate agatha
```

Install PyTorch. We need a version >= 1.4, but different systems will require different cuda library versions. We installed PyTorch using this command:

```
conda install pytorch cudatoolkit=9.2 -c pytorch
```

We use protobufs to help configure aspects of the Agatha pipeline, if you don't already have protoc installed, you can pull it in through conda.

```
conda install -c anaconda protobuf
```

Install Agatha. This comes along with the dependencies necessary to run the pretrained model. Note, we're aware of a pip warning produced by this install method, we're working on providing a easier pip-installable wheel.

```
cd <AGATHA_INSTALL_DIR>
git clone https://github.com/JSybrandt/agatha.git .
pip install -e .
```

Now we can download the 2015 hypothesis prediction subset. Note, at the time of writing, we only provide a 2015 validation version of Agatha. We are in the process of preparing an up-to-date 2020 version. We recommend the tool `gdown` that comes along with Agatha to download our 38.5GB file. If you don't want to use that tool, you

can download the same file from your browser via this link. We recommend you place this somewhere within <AGATHA_INSTALL_DIR>/data.

```
# Remeber where you place your file
cd <AGATHA_DATA_DIR>
# This will place 2015_hypothesis_predictor_512.tar.gz in AGATHA_DATA_DIR
gdown --id 1Tka7zPF0PdG7yvGOGOXuEsAtRLLimXmP
# Unzip the download, creates hypothesis_predictor_512/...
tar -zxvf 2015_hypothesis_predictor_512.tar.gz
```

We can now load the Agatha model in python. After loading, we need to inform the model of where it can find its helper data. By default it looks in the current working directory.

```
# We need to load the pretrained agatha model.
import torch
model = torch.load("<AGATHA_DATA_DIR>/hypothesis_predictor_512/model.pt")

# We need to tell the model abouts its helper data.
model.set_data_root("<AGATHA_DATA_DIR>/hypothesis_predictor_512/")

# We need to setup the internal datastructures around that helper data.
model.init()

# Now we can run queries specifying two umls terms! Note, this process has some
# random smapling involved, so your result might not look exactly like what we
# show here.
# Keywords:
#   Cancer: C0006826
#   Tobacco: C0040329
model.predict_from_terms([("C0006826", "C0040329")])
>>> [0.78358984]

# Kewords:
#   Cancer: C0006826
#   Tobacco: C0040329
model.predict_from_terms([("C0006826", "C0040329")])
>>> [0.78358984]

# If you want to run loads of queries, we recommend first using
# model.init_preload(), and then the following syntax. Note that
# predict_from_terms will automatically compute in batches of size:
# model.hparams.batch_size.
queries = [("C###", "C###"), ("C###", "C###"), ..., ("C###", "C###")]
model = model.eval()
model = model.cuda()
with torch.no_grad():
  predictions = model.predict_from_terms(queries)
```

1.2 Replicate the 2015 Validation Experiments

Provided in `./benchmarks` are the files we use to produce the results found in our paper. Using the 2015 pretrained model, you should be able to replicate these results. This guide focuses on the recommendation experiments, wherein all pairs of elements from among the 100 most popular new predicates per-subdomain are evaluated by the Agatha model. For each of the 20 considered types, we generated all pairs, and removed any pair that is trivially discoverable from within the Agatha semantic graph. The result are a list of predicates in the following json file: `./benchmarks/all_pairs_top_20_types.json`

The json predicate file has the following schema:

```
{
  "<type1>:<type2>": [
    {
      "source": "<source keyword>",
      "target": "<target keyword>",
      "label": 0 or 1
    },
    ...
  ],
  ...
}
```

Here's how to load the pretrained model and evaluate the provided set of predicates:

```
import torch
import json

# Load the pretrained model
model = torch.load("<AGATHA_DATA_DIR>/hypothesis_predictor_512/model.pt")
# Configure the helper data
model.set_data_root("<AGATHA_DATA_DIR>/hypothesis_predictor_512")
# Initialize the model for batch processing
model.init_preload()


# Load the json file
with open("<AGATHA_INSTALL_DIR>/benchmarks/all_pairs_top_20_types") as file:
  types2predicates = json.load(file)

# prepare model
model = model.eval()
model = model.cuda()
with torch.no_grad():

  # Predict ranking criteria for each predicate
  types2predictions = {}
  for typ, predicates in types2predictions.items():
    types2predictions[typ] = model.predict_from_terms([
      (pred["source"], pred["target"])
      for pred in predicates
    ])
```

Note that the order of resulting scores will be the same as the order of the input predicates per-type. Using the `label` field of each predicate, we can then compare how the ranking critera correlates with the true connections `1` and the undisovered connections `0`.

1.3 Installing Agatha for Development

These instructions are useful if you want to customize Agatha, especially if you are also running this system on the Clemson Palmetto Cluster. This guide also assumes that you have already installed `anaconda3`.

Step zero. Get yourself a node made in the last few years with a decent GPU. Currently supported GPU's on palmetto include the `P100` and the `V100`. Recent changes to pytorch are incompatible with older models.

The recommended node request is:

```
qsub -I -l select=5:ncpus=40:mem=365gb:ngpus=2:gpu_model=v100,walltime=72:00:00
```

First, load the following modules:

```
module load gcc/8.3.0          \
            cuDNN/9.2v7.2.1    \
            sqlite/3.21.0      \
            cuda-toolkit/9.2   \
            nccl/2.4.2-1       \
            hdf5/1.10.5        \
            mpc/0.8.1
```

Now follow the above list of installation instructions, beginning with creating a conda environment, through cloning the git repo, and ending with `pip install -e .`.

At this point, we can install all the additional dependencies required to construct the Agatha semantic graph and train the transformer model. To do so, return to the `AGATHA_INSTALL_DIR` and install `requirements.txt`.

```
cd <AGATHA_INSTALL_DIR>
# Installs the developer requirements
pip install -r requirements.txt
```

Now you should be ready to roll! I recommend you create the following file in order to handle all the module loading and preparation.

```
# Remove current modules (if any)
module purge
# Leave current conda env (if any)
conda deactivate
# Load all nessesary palmetto modules
module load gcc/8.3.0 mpc/0.8.1 cuda-toolkit/9.2 cuDNN/9.2v7.2.1 nccl/2.4.2-1 \
            sqlite/3.21.0 hdf5/1.10.5
# Include hdf5, needed to build tools
export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:/software/hdf5/1.10.5/include
# Load python modules
conda activate agatha
```

CHAPTER

TWO

AGATHA

2.1 agatha package

2.1.1 Subpackages

agatha.construct package

Subpackages

agatha.construct.document_parsers package

Submodules

agatha.construct.document_parsers.document_record module

agatha.construct.document_parsers.document_record.**assert_valid_document_record**(*record*)

Return type None

agatha.construct.document_parsers.document_record.**new_document_record**()

Return type Dict[str, Any]

agatha.construct.document_parsers.parse_covid_json module

agatha.construct.document_parsers.parse_covid_json.**json_path_to_record**(*json_path*)

Return type Dict[str, Any]

agatha.construct.document_parsers.parse_pubmed_xml module

agatha.construct.document_parsers.parse_pubmed_xml.**parse_zipped_pubmed_xml**(*xml_path*)
Copies the given xml file to local scratch, and then gets the set of articles, represented by a list of dicts.

Return type List[Dict[str, Any]]

agatha.construct.document_parsers.parse_pubmed_xml.**pubmed_xml_to_record**(*pubmed_elem*)
Given a PubmedArticle element, parse out all the fields we care about. Fields are represented as a dictionary.

Return type Dict[str, Any]

agatha.construct.document_parsers.parse_pubmed_xml.**xml_obj_to_date**(*elem*)

Return type str

agatha.construct.document_parsers.test_parse_covid_json module

agatha.construct.document_parsers.test_parse_covid_json.**get_expected_texts**(*covid_raw*)

agatha.construct.document_parsers.test_parse_covid_json.**load_json**()

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_id_1**()

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_id_2**()

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_success_1**()
Ensures that parsing happens without failure

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_success_2**()
Ensures that parsing happens without failure

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_text_1**()

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_text_2**()

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_title_1**()

agatha.construct.document_parsers.test_parse_covid_json.**test_parse_title_2**()

Module contents

Submodules

agatha.construct.checkpoint module

A singleton responsible for saving and loading dask bags.

agatha.construct.checkpoint.**checkpoint**(*name*, *bag=None*, *verbose=None*, *allow_partial=None*, *halt_after=None*, *textfile=False*, ***compute_kw*)

Stores the contents of the bag as a series of files.

This function takes each partition of the input bag and writes them to files within a directory associated with the input name. The location of each checkpoint directory is dependent on the *ckpt_root* option.

For each optional argument, (other than *bag*) of this function, there is an associated module-level parameter that can be set globally.

The module-level parameter checkpoint_root, set with *set_root* must be set before calling checkpoint.

Usage: checkpoint(name) - returns load opt for checkpoint "name" checkpoint(name, bag) - if ckpt writes bag to ckpt "name" and returns load op if disable() was called, returns the input bag

Parameters

- **name** (str) – The name of the checkpoint directory to lookup or save to
- **bag** (Optional[Bag]) – If set, save this bag. Otherwise, we will require that this checkpoint has already been saved.

- **verbose** (Optional[bool]) – Print helper info. If unspecified, defaults to module-level parameter.
- **allow_partial** (Optional[bool]) – If true, partial files present in an unfinished checkpoint directory will not be overwritten. If false, unfinished checkpoints will be re-computed in full. Defaults to module-level parameter if unset.
- **halt_after** (Optional[str]) – If set to the name of the current checkpoint, the agatha process will stop after computing its contents. This is important for partial pipeline runs, for instance, for computing training data for an ml model.
- **textfile** (bool) – If set, checkpoint will be stored in plaintext format, used to save strings. This results in this function returning *None*.

Return type Optional[Bag]

Returns A dask bag that, if computed, _LOADS_ the specified checkpoint. This means that future operations can depend on the loading of intermediate data, rather than the intermediate computations themselves.

agatha.construct.checkpoint.**ckpt**(*bag_name*, *ckpt_prefix=None*, ***kwargs*)

Simple checkpoint interface

This is syntactic sugar for the most common use case. You can replace ` my_dask_bag = checkpoint("my_dask_bag", my_dask_bag) `

` ckpt("my_dask_bag") `

Calling this function will replace the variable associated with *bag_name* after computing its checkpoint. This means that calling compute on later calls of *bag_name* will load that bag from storage, rather than perform all intermediate computations again.

Parameters

- **bag_name** (str) – The name of a local variable corresponding to a dask bag. This bag will be computed and stored to a checkpoint of the same name. The bag variable will be replaced with a new bag that can be loaded from this checkpoint.
- **ckpt_prefix** (Optional[str]) – If set, the provided string will be prefixed to the bag_name checkpoint. This allows the same variable names to be associated with different checkpoints. For instance, the *document_pipeline* functions create a bag named "sentences" regardless of the set of documents used to create those sentences. By specifying a prefix, different calls to *document_pipeline* can create different checkpoints.

Return type None

agatha.construct.checkpoint.**clear_all_ckpt**()

Return type None

agatha.construct.checkpoint.**clear_ckpt**(*name*)

Return type None

agatha.construct.checkpoint.**clear_halt_point**()

Return type None

agatha.construct.checkpoint.**disable**()

Return type None

agatha.construct.checkpoint.**enable**()

Return type None

agatha.construct.checkpoint.**get_allow_partial**()

Return type bool

agatha.construct.checkpoint.**get_checkpoints_like**(*glob_pattern*)

Return type Set[Path]

agatha.construct.checkpoint.**get_done_file_path**(*name*)

Return type Path

agatha.construct.checkpoint.**get_or_make_ckpt_dir**(*name*)

Return type Path

agatha.construct.checkpoint.**get_root**()

Return type Path

agatha.construct.checkpoint.**get_verbose**()

Return type bool

agatha.construct.checkpoint.**is_ckpt_done**(*name*)

Return type bool

agatha.construct.checkpoint.**set_allow_partial**(*allow*)

Return type None

agatha.construct.checkpoint.**set_halt_point**(*name*)

Return type None

agatha.construct.checkpoint.**set_root**(*ckpt_root*)

Return type None

agatha.construct.checkpoint.**set_verbose**(*is_verbose*)

Return type None

agatha.construct.dask_process_global module

This util is intended to be a universal initializer for all process-specific helper data that is loaded at the start of the construction process. This is only intended for expensive complex structures that must be loaded at startup, and we don't want to reload each function call.

class agatha.construct.dask_process_global.**LocalMockWorker**
Bases: object

class agatha.construct.dask_process_global.**WorkerPreloader**
Bases: object

clear(*worker*)

get(*key*, *worker*)

Return type Any

register(*key*, *init*)
Adds a global object to the preloader

Return type None

setup(*worker*)

teardown(*worker*)

agatha.construct.dask_process_global.**add_global_preloader**(*preloader*, *client=None*)

Return type None

agatha.construct.dask_process_global.**clear**()
Deletes all preloaded data. To be called following a ckpt.

Return type None

agatha.construct.dask_process_global.**get**(*key*)
Gets a value from the global preloader

Return type Any

agatha.construct.dask_process_global.**get_global_preloader**()

agatha.construct.dask_process_global.**get_worker_lock**()

agatha.construct.dask_process_global.**safe_get_worker**()

agatha.construct.document_pipeline module

agatha.construct.document_pipeline.**get_covid_documents**(*config*)

Return type Bag

agatha.construct.document_pipeline.**get_medline_documents**(*config*)

Return type Bag

agatha.construct.document_pipeline.**perform_document_independent_tasks**(*config*, *documents*, *ckpt_prefix*, *semrep_work_dir=None*)
Performs Tasks that don't require communication between documents

Performs all of the document processing operations that are required to happen on each document separately. This is important to separate between different input textual features because this allows us to update/invalidate particular sets of checkpoints faster.

Parameters

- **config** (ConstructConfig) – Constriction Configuration
- **documents** (Bag) – Collection of texts to process
- **ckpt_prefix** (str) – To stop collisions, and to improve caching, each call to this function should have a different prefix indicating the type of the corresponding documents. For instance, calling this with medline documents could get the *medline* prefix.
- **semrep_work_dir** (Optional[Path]) – The location to store semrep intermediate files. Only used if semrep has been installed and configured.

Return type None

agatha.construct.embedding_util module

agatha.construct.embedding_util.**embed_records**(*records*, *batch_size*, *text_field*, *max_sequence_length*, *out_embedding_field='embedding'*)

Introduces an embedding field to each record, indicated the bert embedding of the supplied text field.

Return type Iterable[Dict[str, Any]]

agatha.construct.embedding_util.**get_bert_initializer**(*bert_model*)

The bert_model may be a path or any provided by the transformers module. For instance "bert-base-uncased"

Return type Tuple[str, Callable[[], Any]]

agatha.construct.embedding_util.**get_pretrained_model_initializer**(*name*, *model_class*, *data_dir*, ***model_kwargs*)

Return type Tuple[str, Callable[[], Any]]

agatha.construct.embedding_util.**get_pytorch_device_initalizer**(*disable_gpu*)

Return type Tuple[str, Callable[[], Any]]

agatha.construct.file_util module

agatha.construct.file_util.**copy_to_local_scratch**(*src*, *local_scratch_dir*)

Return type Path

agatha.construct.file_util.**get_part_files**(*dir_path*)

Return type List[Path]

agatha.construct.file_util.**get_random_ascii_str**(*str_len*)

Return type str

agatha.construct.file_util.**is_result_saved**(*path*)

Return type bool

agatha.construct.file_util.**load**(*dir_path*, *allow_failure=False*)

Return type Bag

agatha.construct.file_util.**load_part**(*path*, *allow_failure=False*)

Return type List[Any]

agatha.construct.file_util.**load_random_sample_to_memory**(*data_dir*, *value_sample_rate=1*, *partition_sample_rate=1*, *disable_pbar=False*)

Return type List[Any]

agatha.construct.file_util.**load_to_memory**(*dir_path*, *disable_pbar=False*)

Performs loading right now, without dask

Return type List[Any]

agatha.construct.file_util.**load_value**(*path*)

Loads an arbitrary object.

Return type Any

agatha.construct.file_util.**prep_scratches**(*local_scratch_root*, *shared_scratch_root*, *task_name*)

Return type Tuple[Path, Path]

agatha.construct.file_util.**save**(*bag*, *path*, *keep_partial_result=False*, *textfile=False*)

Return type delayed

agatha.construct.file_util.**save_part**(*part*, *path*, *textfile=False*)
Stores that partition at *path*, returns *path*

Return type Path

agatha.construct.file_util.**save_value**(*value*, *path*)
Saves an arbitrary object.

Return type None

agatha.construct.file_util.**touch_random_unused_file**(*base_dir*, *ext=None*)

Return type Path

agatha.construct.file_util.**wait_for_file_to_appear**(*file_path*, *max_tries=5*)

Return type None

agatha.construct.file_util.**write_done_file**(*parts*, *part_dir*)

Return type Path

agatha.construct.ftp_util module

agatha.construct.ftp_util.**ftp_connect**(*address*, *workdir*)
Connects to a remote FTP server at a specific directory

Return type FTP

agatha.construct.ftp_util.**ftp_download**(*conn*, *remote_name*, *directory*)

Return type Path

agatha.construct.ftp_util.**ftp_download_if_missing**(*conn*, *remote_name*, *directory*)
If the file already exists, skip it.

Return type Path

agatha.construct.ftp_util.**ftp_list_files**(*conn*, *pattern='.*'*)

Return type List[str]

agatha.construct.ftp_util.**ftp_retreive_all**(*conn*, *directory*, *pattern='.*'*, *show_progress=False*)
For each file matching the given pattern, download if not in directory.

Return type List[Path]

agatha.construct.graph_util module

agatha.construct.graph_util.**record_to_bipartite_edges**(*records*, *get_neighbor_keys_fn*, *get_source_key_fn=<function <lambda>>*, *bidirectional=True*)

This function is responsible for extracting edges from records. For example, if you had a bag of records, each containing a set of terms, you might want to get the set of edges between records and terms.

Parameters

- **records** (Bag) – The collection of records we wish to extract edges from.
- **get_neighbor_keys_fn** (Callable[[Dict[str, Any]], Iterable[str]]) – Given a record, return a list of graph keys that are adjacent to the given record
- **get_source_key_fn** (Callable[[Dict[str, Any]], str]) – Given a record, return a graph key that uniquely identifies the root. By default we get the "id" field
- **bidirectional** (bool) – If true, we write record->neighbor and neighbor->record. If false, we only write record->neighbor.

Return type Bag

Returns A bag containing serialized key-value pairs that can be used to create an Sqlite3LookupTable

agatha.construct.knn_util module

agatha.construct.knn_util.**add_points_to_index**(*records*, *init_index_path*, *batch_size*, *output_path*)

Loads an initial index, adds the partition to the index, and writes result

Return type Path

agatha.construct.knn_util.**get_faiss_index_initializer**(*faiss_index_path*, *index_name='final'*)

Return type Tuple[str, Callable[[], Any]]

agatha.construct.knn_util.**merge_index**(*init_index_path*, *partial_idx_paths*, *final_index_path*)

Return type Path

agatha.construct.knn_util.**nearest_neighbors_network_from_index**(*hash_and_embedding*, *hash2name_db*, *batch_size*, *num_neighbors*, *faiss_index_name='final'*, *weight=1.0*)

Applies faiss and runs results through inverted index.

Return type Iterable[str]

agatha.construct.knn_util.**to_hash_and_embedding**(*records*, *id_field='id'*, *embedding_field='embedding'*)

Return type Tuple[ndarray, ndarray]

agatha.construct.knn_util.**train_distributed_knn**(*hash_and_embedding*, *batch_size*, *num_centroids*, *num_probes*, *num_quantizers*, *bits_per_quantizer*, *training_sample_prob*, *shared_scratch_dir*, *final_index_path*, *id_field='id'*, *embedding_field='embedding'*)

Computing all of the embeddings and then performing a KNN is a problem for memory. So, what we need to do instead is compute batches of embeddings, and use them in Faiss to reduce their dimensionality and process the appropriatly.

I'm so sorry this one function has to do so much...

@param hash_and_embedding: bag of hash value and embedding values @param text_field: input text field that we embed. @param id_field: output id field we use to store number hashes @param batch_size: number of sentences per batch @param num_centroids: number of voronoi cells in approx nn @param num_probes: number of cells to consider when querying @param num_quantizers: number of sub-vectors to discritize @param bits_per_quantizer: bits per sub-vector @param shared_scratch_dir: location to store intermediate results. @param training_sample_prob: chance a point is trained on @return The path you can load the resulting FAISS index

Return type Path

agatha.construct.knn_util.**train_initial_index**(*training_data*, *num_centroids*, *num_probes*, *num_quantizers*, *bits_per_quantizer*, *output_path*)

Computes index using method from: https://hal.inria.fr/inria-00514462v2/document

Vector dimensionality must be a multiple of num_quantizers. Input vectors are "chunked" into *num_quantizers* sub-components. Each chunk is reduced to a *bits_per_quantizer* value. Then, the L2 distances between these quantized bits are compared.

For instance, a scibert embedding is 768-dimensional. If num_quantizers=32 and bits_per_quantizer=8, then each vector is split into subcomponents of only 24 values, and these are further reduced to an 8-bit value. The result is that we're only using 1/3 of a bit per value in the input.

When constructing the index, we use quantization along with the L2 metric to perform K-Means, constructing a voronoi diagram over our training data. This allows us to partition the search space in order to make inference faster. *num_centroids* determines the number of voronoi cells a point could be in, while *num_probes* determines the number of nearby cells we consider at query time. So higher centroids means faster and less accurate inference. Higher probes means the opposite, longer and more accurate queries.

According to: https://github.com/facebookresearch/faiss/wiki/Faiss-indexes, we should select #centroids on the order of sqrt(n).

Choosing an index is hard: https://github.com/facebookresearch/faiss/wiki/Index-IO,-index-factory,-cloning-and-hyper-parameter-tuning

Return type None

agatha.construct.ngram_util module

agatha.construct.ngram_util.**get_frequent_ngrams**(*analyzed_sentences*, *max_ngram_length*, *min_ngram_support*, *min_ngram_support_per_partition*, *ngram_sample_rate*, *token_field='tokens'*, *ngram_field='ngrams'*)

Adds a new field containing a list of all mined n-grams. N-grams are tuples of strings such that at least one string is not a stopword. Strings are collected from the lemmas of sentences. To be counted, an ngram must occur in at least *min_ngram_support* sentences.

Return type Bag

agatha.construct.semrep_util module

SemRep Dask Utilities

This module helps run SemRep within the Agatha graph construction pipeline. For this to work, we need to run SemRep on each machine in our cluster, and extract all necessary information as edges.

To run SemRep, you must first start the MetaMap servers for part-of-speech tagging and word-sense disambiguation. These are supplied through MetaMap. Specifically, we are expecting to find *skrmedpostctl* and *wsdserverctl* in the directory specified through *config.semrep.metamap_bin_dir*. Once these servers are started we are free to run *semrep*.

class agatha.construct.semrep_util.**MetaMapServer**(*metamap_install_dir*)

Bases: object

Manages connection to MetaMap

SemRep requires a connection to MetaMap. This means we need to launch the pos_server and wsd_server. This class is responsible for managing that server connection. We anticipate using one server per-worker, meaning this class will be initialized using dask_process_global initializer.

Parameters **metamap_install_dir** (Path) – The install location of MetaMap

running()

start()

Call to start the MetaMap servers, if not already running.

stop()

Stops the MetaMap servers, if running

class agatha.construct.semrep_util.**SemRepRunner**(*semrep_install_dir*, *metamap_server*, *anaphora_resolution=True*, *dysonym_processing=True*, *lexicon_year=2006*, *mm_data_version='USAbase'*, *mm_data_year='2006AA'*, *relaxed_model=True*, *single_line_delim_input_w_id=True*, *use_generic_domain_extensions=False*, *use_generic_domain_modification=False*, *word_sense_disambiguation=True*)

Bases: object

Responsible for running SemRep.

Given a metamap server and additional SemRep Configs, this class actually processes text and generates predicates. All SemRep predicates are copied here and provided through the constructor. All defaults are preserved.

Parameters

- **semrep_install_dir** (Path) – Location where semrep is installed.
- **metamap_server** (*MetaMapServer*) – A connection to the MetaMapServer that enables us to actually run SemRep. We use this to ensure server is running.
- **work_dir** – Location to store intermediate files used to communicate with SemRep.
- **anaphora_resolution** – SemRep Flag
- **dysonym_processing** – SemRep Flag
- **lexicon_year** (int) – The year as an int which we use with MetaMap. Ex: 2020
- **mm_data_version** (str) – Specify which UMLS data version. Ex: USAbase
- **mm_data_year** (str) – Specify UMLS release year. Ex: 2020AA
- **relaxed_model** (bool) – SemRep Flag
- **use_generic_domain_extensions** – SemRep Flag
- **use_generic_domain_modification** – SemRep Flag
- **word_sense_disambiguation** – SemRep Flag

run(*input_path*, *output_path*)

Actually calls SemRep with an input file.

Parameters **input_path** (Path) – The location of the SemRep Input file

Return type None

Returns The path produced by SemRep representing XML output.

class agatha.construct.semrep_util.**UnicodeToAsciiRunner**(*unicode_to_ascii_jar_path*)

Bases: object

Responsible for running the MetaMap unicode to ascii jar

clean_text_for_metamap(*s*)

Metamap has a bunch of stupid rules.

Return type str

agatha.construct.semrep_util.**extract_entities_and_predicates_from_sentences**(*sentence_records*, *semrep_install_dir*, *unicode_to_ascii_jar_pa*
work_dir, *lexicon_year*, *mm_data_year*, *mm_data_version*)

Runs each sentence through SemRep. Identifies Predicates and Entities

Requires get_metamap_server_initializer added to dask_process_global.

Parameters

- **sentence_records** (Bag) – Each record needs *id* and *sent_text*.
- **work_dir** (Path) – A directory visible to all workers where SemRep intermediate files will be stored.
- **semrep_install_dir** (Path) – The path where semrep was installed.

Return type Bag

Returns One record per input sentence, where *id* of the new record matches the input. However, returned records will only have *entites* and *predicates*

agatha.construct.semrep_util.**get_metamap_server_initializer**(*metamap_install_dir*)

Return type Tuple[str, Callable[[], Any]]

agatha.construct.semrep_util.**get_paths**(*semrep_install_dir=None*, *metamap_install_dir=None*)

Looks up all of the necessary files needed to run SemRep.

This function identifies the binaries and libraries needed to run SemRep. Additionally, this function asserts that all the needed files are actually present.

This function will find: *skrmedpostctl*: Metamap's SKR/Medpost Part-of-Speech Tagger Server *wsdserverctl*: Metamap's Word Sense Disambiguation (WSD) Server *SEMREPrun.v**: The preamble needed to run SemRep *semrep.v*.BINARY.Linux*: The binary used to run SemRep *lib*: The Java libraries in SemRep

If only one or the other semrep_install_dir or metamap_install_dir is specified, then only that components paths will be returned.

Parameters

- **semrep_install_dir** (Optional[Path]) – The install location of SemRep. Named *public_semrep* by default.
- **metamap_install_dir** (Optional[Path]) – The install location of MetaMap. Named *public_mm* my default.

Return type Dict[str, Path]

Returns A dictionary of names and associated paths. If a name ends in *_path* then it has been asserted *is_file()*. If name ends in *_dir* it has been asserted *is_dir()*.

agatha.construct.semrep_util.**semrep_xml_to_records**(*xml_path*)

Parses SemRep XML records to produce Predicate Records

This parses SemRep XML output, generated by SemRep v1.8 via the *–xml_output_format* flag. Take a look [here][1] to get more details on the XML spec. Additional details below. We specifically focus on parsing XML records produced by the SemRepRunner.

XML Format Summary: The XML file starts with an overarching SemRepAnnotation object, containing multiple *Document* records, one per input text. These documents contain identified UMLS terms (*Document* > *Utterance* > *Entity*) and predicates (*Document* > *Utterance* > *Predication*). One document may have multiple utterances.

Parameters **xml_path** (Path) – Location of XML file to parse.

Return type List[Dict[str, Any]]

Returns A list of python dicts wherein each corresponds to a detected predicate.

[1]:https://semrep.nlm.nih.gov/SemRep.v1.8_XML_output_desc.html

agatha.construct.semrep_util.**sentences_to_semrep_input**(*records*, *unicode_to_ascii_jar_path*)

Processes Sentence Records for SemRep Input

The SemRepRunner, with the default single_line_delim_input_w_id flag set, expects input in the form: ``` id1|Sentence 1 id2|Sentence 2

...

```

This function converts Agatha sentence records, containing the *sent_text* and *id* fields into the single_line_delim_input_w_id format. Because each sentence must occur on its own line, this function will replace newline characters with spaces in output.

Recommend Usage:

```python3 sentences.map_partitions(

sentences_to_semrep_input, unicode_to_ascii_jar_path,

).to_textfiles(...) ```

**Parameters**

- **records** (Iterable[Dict[str, Any]]) – Sentence records, each containing *sent_text* and *id*
- **unicode_to_ascii_jar_path** (Path) – The location of the metamap-provided jar

**Return type** List[str]

### agatha.construct.test_checkpoint module

agatha.construct.test_checkpoint.**setup_done_checkpoints**(*root_name*)

**Return type** None

agatha.construct.test_checkpoint.**test_clear_ckpt**()

agatha.construct.test_checkpoint.**test_default_none_ckpt_root**()

agatha.construct.test_checkpoint.**test_get_checkpoints_like**()

agatha.construct.test_checkpoint.**test_get_checkpoints_like_complete**()

agatha.construct.test_checkpoint.**test_get_or_make_ckpt_dir**()

agatha.construct.test_checkpoint.**test_set_root**()

agatha.construct.test_checkpoint.**test_setup_done_checkpoints**()

### agatha.construct.test_file_util module

agatha.construct.test_file_util.**async_touch_after**(*file_path*, *wait_before*)

Spawns a process to touch a file

**Return type** None

agatha.construct.test_file_util.**test_async_touch_after**()

agatha.construct.test_file_util.**test_wait_for_file_to_appear_exists**()

agatha.construct.test_file_util.**test_wait_for_file_to_appear_not_exists**()

### agatha.construct.test_semrep_util module

agatha.construct.test_semrep_util.**test_extract_entitites_and_predicates_with_dask**()

agatha.construct.test_semrep_util.**test_get_all_paths**()
Tests that getting semrep paths gets all needed paths

agatha.construct.test_semrep_util.**test_get_metamap_paths**()
Tests that getting semrep paths gets all needed paths

agatha.construct.test_semrep_util.**test_get_semrep_paths_fails**()
Tests that if you give semrep paths bad install locations, it fails

agatha.construct.test_semrep_util.**test_metamap_server**()
Tests that we can actually run metamap

agatha.construct.test_semrep_util.**test_parse_semrep_end_to_end**()

agatha.construct.test_semrep_util.**test_parse_semrep_end_to_end_difficult**()

agatha.construct.test_semrep_util.**test_parse_semrep_xml_entity**()

agatha.construct.test_semrep_util.**test_parse_semrep_xml_predication**()

agatha.construct.test_semrep_util.**test_run_semrep**()

agatha.construct.test_semrep_util.**test_run_semrep_covid**()

agatha.construct.test_semrep_util.**test_semrep_fails_with_bad_sentence**()
We ran into a problem with the following abstract:

https://pubmed.ncbi.nlm.nih.gov/3624238/

Specifically, the component that includes a list of names:

(Samanta, H., Engel, D. A., Chao, H. M., Thakur, A., Garcia-Blanco, M. A., and Lengyel, P. (1986) J. Biol. Chem. 261, 11849-11858).

Was split into these sentences:

1: (Samanta, H., Engel, D. 2: A., Chao, H. . . .

The sentence *A., Chao, H.* causes an error due to an unforseen exception within semrep.

This problematic abstract is represented here and processed in the same way as in the typical dask pipeline.

agatha.construct.test_semrep_util.**test_semrep_id_to_agatha_sentence_id**()

agatha.construct.test_semrep_util.**test_semrep_id_to_agatha_sentence_id_weird_id**()

agatha.construct.test_semrep_util.**test_semrep_paths**()
Tests that if we just need the semrep paths, we can get those

agatha.construct.test_semrep_util.**test_semrep_xml_to_records**()
Ensures that parsing xml files happens without error

agatha.construct.test_semrep_util.**test_sentence_to_semrep_input**()

agatha.construct.test_semrep_util.**test_sentence_to_semrep_input_filter_newline**()

agatha.construct.test_semrep_util.**test_sentence_to_semrep_input_filter_single_quote**()

agatha.construct.test_semrep_util.**test_sentence_to_semrep_input_filter_unicode**()

agatha.construct.test_semrep_util.**test_unicode_to_ascii**()

### agatha.construct.text_util module

agatha.construct.text_util.**add_bow_to_analyzed_sentence**(*records*, *bow_field='bow'*, *token_field='tokens'*, *entity_field='entities'*, *mesh_heading_field='mesh_headings'*, *ngram_field='ngrams'*)

**Return type** Dict[str, Any]

agatha.construct.text_util.**analyze_sentences**(*records*, *text_field*, *token_field='tokens'*, *entity_field='entities'*)

Parses the text fields of all records using SciSpacy. Requires that text_util:nlp and text_util:stopwords have both been loaded into dask_process_global.

@param records: A partition of records to parse, each must contain *text_field* @param text_field: The name of the field we wish to parse. @param token_field: The output field for all basic tokens. These are sub-records containing information such as POS tag and lemma. @param entity_field: The output field for all entities, which are multi-token phrases. @return a list of records with token and entity fields

**Return type** Iterable[Dict[str, Any]]

agatha.construct.text_util.**entity_to_id**(*entity*, *sentence*, *token_field='tokens'*)

**Return type** str

agatha.construct.text_util.**get_adjacent_sentences**(*sentence_record*)

Given the i'th sentence, return the keys for sentence i-1 and i+1 if they exist.

**Return type** Set[str]

agatha.construct.text_util.**get_entity_keys**(*sentence_record*)

**Return type** List[str]

agatha.construct.text_util.**get_entity_text**(*entity*, *sentence*, *token_field='tokens'*)

**Return type** str

agatha.construct.text_util.**get_interesting_token_keys**(*sentence_record*)

**Return type** List[str]

agatha.construct.text_util.**get_mesh_keys**(*sentence_record*)

**Return type** List[str]

agatha.construct.text_util.**get_ngram_keys**(*sentence_record*)

**Return type** List[str]

agatha.construct.text_util.**get_scispacy_initalizer**(*scispacy_version*)

**Return type** Tuple[str, Callable[[], Any]]

agatha.construct.text_util.**get_sentence_id**(*pmid*, *version*, *sent_idx*)

**Return type** str

agatha.construct.text_util.**get_stopwordlist_initializer**(*stopword_path*)

**Return type** Tuple[str, Callable[[], Any]]

agatha.construct.text_util.**mesh_to_id**(*mesh_code*)

**Return type** str

agatha.construct.text_util.**ngram_to_id**(*ngram_text*)

**Return type** str

agatha.construct.text_util.**sentence_to_id**(*sent*)

**Return type** str

agatha.construct.text_util.**split_sentences**(*records*, *text_data_field='text_data'*, *id_field='id'*, *min_sentence_len=None*, *max_sentence_len=None*)

Splits a document into its collection of sentences. In order of text field elements, we split sentences and create new elements for the result. All fields from the original document, as well as the text field (minus the actual text itself) are copied over.

If min/max sentence len are specified, we do NOT consider sentences that fail to match the range.

*id_field* will be set with {SENTENCE_TYPE}:{pmid}:{version}:{sent_idx}

For instance:

**{** "status": "Published", "umls": ["C123", "C456"], "text_fields": [{

"text": "Title 1", "type": "title"

**}, {** "text": "This is an abstract. This is another sentence.", "type": "abstract:raw",

}]

}

becomes:

**[{** "status": "Published", "umls": ["C123", "C456"], "sent_text": "Title 1", "sent_type": "title", "sent_idx": 0, "sent_total": 3, },{ "status": "Published", "umls": ["C123", "C456"], "sent_text": "This is an abstract.", "sent_type": "abstract:raw", "sent_idx": 1, "sent_total": 3, },{ "status": "Published", "umls": ["C123", "C456"], "sent_text": "This is another sentence.", "sent_type": "abstract:raw", "sent_idx": 2, "sent_total": 3,

}]

**Return type** List[Dict[str, Any]]

agatha.construct.text_util.**token_to_id**(*token*)

**Return type** str

### Module contents

### agatha.ml package

### Subpackages

### agatha.ml.abstract_generator package

### Submodules

### agatha.ml.abstract_generator.abstract_generator module

### agatha.ml.abstract_generator.datasets module

agatha.ml.abstract_generator.datasets.**collate_encoded_abstracts**(*batch*, *key_subset=None*)

**Return type** Dict[str, LongTensor]

agatha.ml.abstract_generator.datasets.**shift_text_features_for_training**(*batch*)

**Return type** Tuple[Dict[str, LongTensor], Dict[str, LongTensor]]

### agatha.ml.abstract_generator.generation_util module

### agatha.ml.abstract_generator.misc_util module

**class** agatha.ml.abstract_generator.misc_util.**HashedIndex**(*max_index*)

Bases: object

This class acts as a dict that maps items to a fixed range of values. Items must be convertible to strings. Value -> Idx idx -> Set of values

**add**(*elem*)

**Return type** None

**get_elements**(*idx*)

**Return type** Set[Any]

**get_index**(*elem*)

**Return type** int

**has_element**(*elem*)

**Return type** bool

**has_index**(*idx*)

**Return type** bool

**class** agatha.ml.abstract_generator.misc_util.**OrderedIndex**

Bases: object

Same exact interface as hashed index, without the hashing

**add**(*elem*)

**Return type** None

**get_elements**(*idx*)

**Return type** Set[Any]

**get_index**(*elem*)

**Return type** int

**has_element**(*elem*)

**Return type** bool

**has_index**(*idx*)

**Return type** bool

agatha.ml.abstract_generator.misc_util.**items_to_hashed_index**(*collection*, *max_index*)

**Return type** *HashedIndex*

agatha.ml.abstract_generator.misc_util.**items_to_ordered_index**(*collection*)

**Return type** *OrderedIndex*

### agatha.ml.abstract_generator.path_util module

### agatha.ml.abstract_generator.prep_training_data module

### agatha.ml.abstract_generator.tokenizer module

**class** agatha.ml.abstract_generator.tokenizer.**AbstractGeneratorTokenizer**(*tokenizer_model_path*, *extra_data_path*, *lower-er-case*)

Bases: object

**decode_dep**(*idx*)

**Return type** str

**decode_entity_label**(*idx*)

**Return type** str

**decode_mesh**(*idx*)

**Return type** str

**decode_pos**(*idx*)

**Return type** str

**decode_text**(*ids*)

**Return type** str

**decode_year**(*idx*)

**Return type** int

**encode_dep**(*dep*)

**Return type** int

**encode_entity_label**(*entity_label*)

**Return type** int

**encode_for_generation**(*initial_text=None*, *year=None*, *mesh_terms=None*, *allow_unknown_terms=False*)

Given initial text and condition data, produce model_in. Intended use:

model = … model.forward(**model.tokenizer.encode_for_generation(

initial_text, year, terms

))

**Return type** Dict[str, LongTensor]

**encode_mesh**(*mesh*)

**Return type** int

**encode_pos**(*pos*)

**Return type** int

**encode_sentence**(*sentence*, *is_first=False*, *is_last=False*)

**Return type** Dict[str, List[int]]

**encode_year**(*year*)

**Return type** int

**len_dep**()

**Return type** int

**len_entity_label**()

**Return type** int

**len_mesh**()

**Return type** int

**len_pos**()

**Return type** int

**len_text**()

**Return type** int

**len_year**()

**Return type** int

**simple_encode_text**(*text*)

**Return type** List[int]

agatha.ml.abstract_generator.tokenizer.**get_current_year**()

### Module contents

### agatha.ml.gpt2_finetune package

### Submodules

### agatha.ml.gpt2_finetune.gpt2_finetune module

agatha.ml.gpt2_finetune.gpt2_finetune.**abstract_record_to_string**(*abstract*)

**Return type** str

agatha.ml.gpt2_finetune.gpt2_finetune.**collate_token_batch**(*tokens*, *include_labels=True*, *device=None*)

**Return type** Dict[str, Any]

agatha.ml.gpt2_finetune.gpt2_finetune.**weighted_index_sample**(*weights*, *omit_small_terms=False*)

Performs weighted sample of weights. Returns index. :type weights: FloatTensor :param weights: len <vocab_size> :type omit_small_terms: bool :param omit_small_terms: If words have a weighted probability less than

*1/len(weights)* they will not be considered.

**Return type** List[int]

**Returns** weighted sample index for each input in batch

### Module contents

### agatha.ml.hypothesis_predictor package

### Submodules

### agatha.ml.hypothesis_predictor.hypothesis_predictor module

### agatha.ml.hypothesis_predictor.predicate_util module

**class** agatha.ml.hypothesis_predictor.predicate_util.**NegativePredicateGenerator**(*coded_terms*, *graph*)

Bases: object

**generate**()

**class** agatha.ml.hypothesis_predictor.predicate_util.**PredicateEmbeddings**(*subj: numpy.array*, *obj: numpy.array*, *subj_neigh: List[numpy.array]*, *obj_neigh: List[numpy.array]*)

Bases: object

**obj: np.array = None**

**obj_neigh: List[np.array] = None**

**subj: np.array = None**

**subj_neigh: List[np.array] = None**

**class** agatha.ml.hypothesis_predictor.predicate_util.**PredicateObservationGenerator**(*graph*, *embeddings*, *neighbor_sample_*

Bases: object

Converts predicate names to predicate observations

**class** agatha.ml.hypothesis_predictor.predicate_util.**PredicateScrambleObservationGenerator**(*p*

Bases: *agatha.ml.hypothesis_predictor.predicate_util.PredicateObservationGenerator*

Same as above, but the neighborhood comes from randomly selected predicates

agatha.ml.hypothesis_predictor.predicate_util.**clean_coded_term**(*term*)
If term is not formatted as an agatha coded term key, produces a coded term key. Otherwise, just returns the term.

**Return type** str

agatha.ml.hypothesis_predictor.predicate_util.**collate_predicate_embeddings**(*predicate_embeddings*)

agatha.ml.hypothesis_predictor.predicate_util.**collate_predicate_training_examples**(*examples*)
Takes a list of results from PredicateExampleDataset and produces tensors for input into the agatha training model.

**Return type** Dict[str, Any]

agatha.ml.hypothesis_predictor.predicate_util.**is_valid_predicate_name**(*predicate_name*)

**Return type** bool

agatha.ml.hypothesis_predictor.predicate_util.**parse_predicate_name**(*predicate_name*)
Parses subject and object from predicate name strings.

Predicate names are formatted strings that follow this convention: p:{subj}:{verb}:{obj}. This function extracts the subject and object and returns coded-term names in the form: m:{entity}. Will raise an exception if the predicate name is improperly formatted.

**Parameters** **predicate_name** (str) – Predicate name in form p:{subj}:{verb}:{obj}.

**Return type** Tuple[str, str]

**Returns** The subject and object formulated as coded-term names.

agatha.ml.hypothesis_predictor.predicate_util.**to_predicate_name**(*subj*, *obj*, *verb='unknown'*)
Converts two names into a predicate of form p:t1:verb:t2

Assumes that terms are correct Agatha graph keys. This means that we expect input terms in the form of m:____. Allows for a custom verb type, but defaults to unknown. Output will always be set to lowercase.

Example usage:

```
` to_predicate_name(m:c1, m:c2) > p:c1:unknown:c2 to_predicate_name(m:c1,
m:c2, "treats") > p:c1:treats:c2 to_predicate_name(m:c1, m:c2, "TREATS") >
p:c1:treats:c2 `
```

Parameters

- **subj** (str) – Subject term. In the form of "m:_____"
- **obj** (str) – Object term. In the form of "m:_____"
- **verb** (str) – Optional verb term for resulting predicate.

Return type str

Returns Properly formatted predicate containing subject and object. Verb type will be set to "UNKNOWN"

agatha.ml.hypothesis_predictor.test_predicate_util module

agatha.ml.hypothesis_predictor.test_predicate_util.**test_clean_coded_term**()

agatha.ml.hypothesis_predictor.test_predicate_util.**test_clean_coded_term_lower**()

agatha.ml.hypothesis_predictor.test_predicate_util.**test_clean_coded_term_passthrough**()

agatha.ml.hypothesis_predictor.test_predicate_util.**test_clean_coded_term_passthrough_lower**

agatha.ml.hypothesis_predictor.test_predicate_util.**test_is_valid_predicate_name**()

Module contents

agatha.ml.util package

Submodules

agatha.ml.util.embedding_lookup module

class agatha.ml.util.embedding_lookup.**EmbeddingLookupTable**(*embedding_dir*, *entity_db*, *disable_cache=False*)

Bases: object

clear_cache()

Return type None

disable_cache()

Return type None

enable_cache()

Return type None

is_preloaded()

the entity index is loaded and all paths have been loaded

Return type bool

keys()

Return type Set[str]

preload()

Return type None

agatha.ml.util.embedding_lookup.**parse_embedding_path**(*path*)

Given a path to an embedding hdf5 file with a name like: embeddings_s_99.v5.h5 return (entity_type, partition_index)

Return type Tuple[str, int]

agatha.ml.util.hparam_util module

agatha.ml.util.hparam_util.**remove_paths_from_namespace**(*hparams*)

Removes variables from the namespace that ends in _db, _dir, or _path

The model is going to include all hparams in the checkpoint. This is a problem for path variables that are needed during training, but are not wanted in the release of the model. For instance, during training we are going to need to tell the model about the embeddings and helper database locations, as well as where to save the model. These paths are machine specific. When we release the model, or even when we start to move files around, these paths will not be consistent.

Parameters **hparams** (Namespace) – The result of calling parse_args.

Returns A copy of hparams with no variables ending in _db, _dir, or _path. Also removes any variables of type Path.

agatha.ml.util.kv_store_dataset module

agatha.ml.util.kv_store_dataset.**get_sqlite_files**(*base_dir*)

Return type List[Path]

agatha.ml.util.lamb_optimizer module

Lamb optimizer.

agatha.ml.util.sqlite3_dataset module

agatha.ml.util.test_embedding_lookup module

agatha.ml.util.test_embedding_lookup.**assert_table_contains_embeddings**(*actual=typing.Dict[str, typing.List[int]], expected=<class 'agatha.ml.util.embedding_loo*

Return type None

agatha.ml.util.test_embedding_lookup.**assert_writable**(*path*)

Return type None

agatha.ml.util.test_embedding_lookup.**make_embedding_hdf5s**(*part2embs*, *embedding_dir*)

This function creates an embedding hdf5 file for test purposes.

Return type None

agatha.ml.util.test_embedding_lookup.**make_entity_lookup_table**(*part2names*, *test_dir*)

Writes embedding location database

Return type Path

agatha.ml.util.test_embedding_lookup.**setup_embedding_lookup_data**(*name2vec*, *test_name*, *num_parts*, *test_root_dir=PosixPath('/tmp')*)

Creates an embedding hdf5 file and an entity sqlite3 database for testing

Parameters

- **name2vec** (Dict[str, List[int]]) – name2vec[x] = embedding of x
- **test_name** (str) – A unique name for this test
- **num_parts** (int) – The number of partitions to split this dataset among.

Return type Tuple[Path, Path]

Returns embedding_dir, entity_db_path You can run EmbeddingLookupTable(*setup_embedding_lookup_data(...))

agatha.ml.util.test_embedding_lookup.**test_embedding_keys**()

agatha.ml.util.test_embedding_lookup.**test_setup_lookup_data**()

agatha.ml.util.test_embedding_lookup.**test_setup_lookup_data_two_parts**()

agatha.ml.util.test_embedding_lookup.**test_typical_embedding_lookup**()

agatha.ml.util.test_sqlite3_dataset module

agatha.ml.util.test_sqlite3_dataset.**test_getitem**()

agatha.ml.util.test_sqlite3_dataset.**test_len**()

agatha.ml.util.test_sqlite3_dataset.**test_subset**()

Module contents

Submodules

agatha.ml.model_summary module

agatha.ml.model_summary.**print_model_summary**(*model*)

Return type None

agatha.ml.module module

Module contents

agatha.topic_query package

Submodules

agatha.topic_query.aux_result_data module

This module is responsible for adding auxiliary helper data to the result proto

agatha.topic_query.aux_result_data.**add_topical_network**(*result*, *topic_model*, *dictionary*, *graph_db*, *bow_db*)
Adds the topical_network field to the result proto. Creates this network by the weighted jacquard of topics.

The source and target words are going to be assigned indices -1 and -2.

Return type None

agatha.topic_query.aux_result_data.**estimate_plaintext_from_graph_key**(*graph_key*, *graph_db*, *bow_db*, *num_sent_to_check=100*)
Given a graph key, get the most likely plaintext word associated with it. For instance, given "l:noun:cancer" or "m:d009369" we should get something like "cancer"

Return type Optional[str]

agatha.topic_query.bow_util module

agatha.topic_query.bow_util.**filter_words**(*keys*, *text_corpus*, *stopwords*)

Return type List[List[str]]

agatha.topic_query.bow_util.**get_document_frequencies**(*text_documents*)
Returns the document occurrence rate for words across documents

Return type Dict[str, int]

agatha.topic_query.path_util module

agatha.topic_query.path_util.**clear_node_attribute**(*graph*, *attribute*, *reinitialize=None*)
Replaces the attribute with the reinitialize, or removes the attribute entirely if None specified

Return type None

agatha.topic_query.path_util.**get_element_with_min_criteria**(*data*, *criteria*)

Return type Any

agatha.topic_query.path_util.**get_nearby_nodes**(*graph_index*, *source*, *max_result_size*, *max_degree*, *key_type=None*, *cached_graph=None*, *disable_pbar=False*)
Returns a collection of entity names corresponding to the nearest neighbors of *source*. This will extend to multi-hop neighbors. @param db_client: Connection to Redis server. @param source: Source node, must be of graph type. @param max_result_size: only return the closest X neighbors. @param key_type: If supplied, only return nodes of the given type. @return list of graph keys, closest to furthest

Return type List[str]

agatha.topic_query.path_util.**get_shortest_path**(*graph_index*, *source*, *target*, *max_degree*, *disable_pbar=False*)
Gets the exact shortest path between two nodes in the network. This method runs a bidirectional search with an amortized download. At a high level, we are storing each node's distance to both the source and target at the same time. Each visit of a node leads us to identify any neighbors with shorter source / target distances. We know we're done when we uncover a node with a tightened path to both source and target.

Return type Tuple[Optional[List[str]], Graph]

agatha.topic_query.path_util.**recover_shortest_path_from_tight_edges**(*graph*, *bridge_node*)

Return type List[str]

agatha.topic_query.test_bow_util module

agatha.topic_query.test_bow_util.**test_filter_words**()

agatha.topic_query.test_path_util module

agatha.topic_query.test_path_util.**node_sets_equal**(*g1*, *g2*)

Return type bool

agatha.topic_query.test_path_util.**test_delete_attribute**()

agatha.topic_query.test_path_util.**test_replace_attribute**()

Module contents

agatha.util package

Submodules

agatha.util.entity_types module

agatha.util.entity_types.**is_data_bank_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_entity_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_gene_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_graph_key**(*name*)

Return type bool

agatha.util.entity_types.**is_lemma_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_mesh_term_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_ngram_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_predicate_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_sentence_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**is_type**(*type_key*, *name*)
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** (str) – Unsure name we're querying
- **type_key** (str) – Single character type, such as one of the strings in this module.

Return type bool

agatha.util.entity_types.**is_umls_term_type**(*name: str*) → bool
True if name is an appropriately formatted key of the specified type.

Names should be in the form "{type_key}:{name}"

Parameters

- **name** – Unsure name we're querying
- **type_key** – Single character type, such as one of the strings in this module.

agatha.util.entity_types.**to_graph_key**(*name*, *key*)

Return type str

agatha.util.misc_util module

agatha.util.misc_util.**flatten_list**(*list_of_lists*)

Return type List[Any]

agatha.util.misc_util.**generator_to_list**(**args*, *gen_fn=None*, ***kwargs*)

agatha.util.misc_util.**hash_str_to_int**(*s*)

agatha.util.misc_util.**hash_str_to_int32**(*s*)

agatha.util.misc_util.**hash_str_to_int64**(*s*)

agatha.util.misc_util.**iter_to_batches**(*iterable*, *batch_size*)
Chunks the input iterable into fixed-sized batches. .. rubric:: Example

```python3 list(iter_to_batches(range(10), 3)) [

[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]

agatha.util.misc_util.**merge_counts**(*key_to_doc_count_1*, *key_to_doc_count_2=None*)
Adds up counts from two dicts

**Return type** Dict[str, Any]

### agatha.util.proto_util module

This module contains the functions necessary to load and manipulate proto objects. Key components of this module include the load_proto function, that wraps multiple proto parsers, as well as parse_args_into_proto, which loads and augments a proto from the command line.

agatha.util.proto_util.**get_field**(*proto_obj*, *field*)
Looks up a message or field from the supplied proto in the same way that getattr might. However, this function is smart enough to handle nested messages in terms of "a.b.c"

**Return type** Any

agatha.util.proto_util.**get_full_field_names**(*proto_obj*)
Lists all field names in all nested messages in the given proto. For instance: : message Foo { : optional string str_1 = 1; : optional string str_2 = 2; : } : message Bar { : optional Foo foo = 1; : optional int32 num = 2; : } get_full_field_names(Foo()) contains [str_1, str_2] get_full_field_names(Bar()) contains [num, foo.str_1, foo.str_2]

**Return type** List[str]
```

agatha.util.proto_util.**load_json_as_proto**(*path*, *proto_obj*)
Interprets *path* as a plaintext json file. Reads the data into *proto_obj* and returns a reference with the result.

Return type ~ProtoObj

agatha.util.proto_util.**load_proto**(*path*, *proto_obj*)
Attempts to parse the provided file using all available parsers. Raises exception if unavailable.

Return type ~ProtoObj

agatha.util.proto_util.**load_serialized_pb_as_proto**(*path*, *proto_obj*)
Interprets the file *path* as a serialized proto. Reads data into *proto_obj*. Returns a reference to the same *proto_obj*

Return type ~ProtoObj

agatha.util.proto_util.**load_text_as_proto**(*path*, *proto_obj*)
Interprets *path* as a plaintext proto file. Reads the data into *proto_obj* and returns a reference with the result.

Return type ~ProtoObj

agatha.util.proto_util.**parse_args_to_config_proto**(*config_proto*)
Combines the above steps to replace parse_args.

Return type None

agatha.util.proto_util.**set_field**(*proto_obj*, *field*, *val*)

Return type None

agatha.util.proto_util.**setup_parser_with_proto**(*config_proto*)
This class parses command line arguments into the config_proto. The idea is to have sensible defaults at all levels. Default Levels:

- Written in proto definition
- Written in config file
- Written on command line

Return type ArgumentParser

agatha.util.proto_util.**transfer_args_to_proto**(*args*, *config_proto*)
Writes the fields from the args to the config proto.

Return type None

agatha.util.semmeddb_util module

agatha.util.semmeddb_util.**assert_datestr**(*date*)

Return type None

agatha.util.semmeddb_util.**earliest_occurances**(*semmeddb*)

Return type Dict[str, str]

agatha.util.semmeddb_util.**filter_by_date**(*semmeddb*, *cut_date*)

Return type Iterable[Dict[str, str]]

agatha.util.semmeddb_util.**parse**(*semmeddb_csv_path*, *silent_tqdm=False*)

Return type Iterable[Dict[str, str]]

agatha.util.semmeddb_util.**predicate_to_key**(*predicate*)

Return type str

agatha.util.semmeddb_util.**split_multi_term_predicate**(*predicate*)

Return type Iterable[Dict[str, str]]

agatha.util.sqlite3_lookup module

class agatha.util.sqlite3_lookup.**Sqlite3Bow**(*db_path*, *table_name='sentences'*, *key_column_name='id'*, *value_column_name='bow'*, ***kwargs*)

Bases: *agatha.util.sqlite3_lookup.Sqlite3LookupTable*

For backwards compatibility, Sqlite3Bow allows for alternate default table, key, and value names. However, newer tables following the default Sqlite3LookupTable schema will still work.

class agatha.util.sqlite3_lookup.**Sqlite3Graph**(*db_path*, *table_name='graph'*, *key_column_name='node'*, *value_column_name='neighbors'*, ***kwargs*)

Bases: *agatha.util.sqlite3_lookup.Sqlite3LookupTable*

For backwards compatibility, Sqlite3Graph allows for alternate default table, key, and value names. However, newer tables following the default Sqlite3LookupTable schema will still work.

class agatha.util.sqlite3_lookup.**Sqlite3LookupTable**(*db_path*, *table_name='lookup_table'*, *key_column_name='key'*, *value_column_name='value'*, *disable_cache=False*)

Bases: object

Dict-like interface for Sqlite3 key-value tables

Assumes that the provided sqlite3 path has a table containing string keys and json-encoded string values. By default, the table name is *lookup_table*, with columns *key* and *value*.

This interface is pickle-able, and provides caching and preloading. Note that instances of this object that are recovered from pickles will _NOT_ retain the preloading or caching information from the original.

Parameters

- **db_path** (Path) – The file-system location of the Sqlite3 file.
- **table_name** (str) – The sql table name to find within *db_path*.
- **key_column_name** (str) – The string column of *table_name*. Performance of the Sqlite3LookupTable will depend on whether an index has been created on *key_column_name*.
- **value_column_name** (str) – The json-encoded string column of *table_name*
- **disable_cache** (bool) – If set, objects resulted from json parsing will not be cached

clear_cache()

Removes contents of internal cache

Return type None

connected()

True if the database connection has been made.

Return type bool

disable_cache()
Disables the use of internal cache

Return type None

enable_cache()
Enables the use of internal cache

Return type None

is_preloaded()
True if database has been loaded to memory.

Return type bool

iterate(*where=None*)
Returns an iterator to the underlying database. If *where* is specified, returned rows will be conditioned. Note, when writing a *where* clause that columns are *key* and *value*

keys()
Get all keys from the Sqlite3 Table.

Recalls _all_ keys from the connected database. This operation may be slow or even infeasible for larger tables.

Return type Set[str]

Returns The set of all keys from the connected database.

preload()
Copies the database to memory.

This is done by dumping the contents of disk into ram, and _does **not**_ perform any json parsing. This improves performance because now sqlite3 calls do not have to travel to storage.

Return type None

agatha.util.sqlite3_lookup.**compile_kv_json_dir_to_sqlite3**(*json_data_dir*, *result_database_path*, *agatha_install_path*, *merge_duplicates*, *verbose*)
Merges all key/value json entries into an indexed sqlite3 table

This function assumes that *json_dir* contains many *.json files. Each file should contain one json object per line. Each object should contain a "key" and a "value" field. This function will use the c++ *create_lookup_table* by executing a subprocess.

Parameters

- **json_data_dir** (Path) – The location containing *.jso. files.
- **result_database_path** (Path) – The location to store the result sqlite3 db.
- **agatha_install_path** (Path) – The location containing the "tools" directory, where *create_lookup_table* has been built.
- **merge_duplicates** (bool) – The create_lookup_table utility has two modes. If merge_duplicates is False, then we assume there are no key collisions and each value is stored as-is. If True, then we combine values associated with duplicate keys into arrays of unique elements.
- **verbose** (bool) – If set, print intermediate output of create_lookup_table.

Return type None

agatha.util.sqlite3_lookup.**create_lookup_table**(*key_value_records*, *result_database_path*, *intermediate_data_dir*, *agatha_install_path*, *merge_duplicates=False*, *verbose=False*)

Creates an Sqlite3 table compatible with Sqlite3LookupTable

Each element of the key_value_records bag is converted to json and written to disk. Then, one machine calls the *create_lookup_table* tool in order to index all records into an Sqlite3LookupTable compatible database. Warning, if used in a distributed setting, the master node will be the one to call the *create_lookup_table* utility.

key_value_records: A dask bag containing dicts. Each dict should have a "key" and a "value" field.

result_database_path: The location to write the Sqlite3 file. intermediate_data_dir: The location to write intermediate json text files.

Warning, if any json files exist beforehand, they will be erased.

agatha_install_path: The root of Agatha, wherein the *tools* directory can be located.

merge_duplicates: If set, *create_lookup_table* will perform the more expensive operation of combining distinct values associated with the same key.

verbose: If set, the *create_lookup_table* utility will print intermediate output.

Return type None

agatha.util.sqlite3_lookup.**export_key_value_records**(*key_value_records*, *export_dir*)

Converts a Dask bag of Dicts into a collection of json files.

In order to create a lookup table, we must first export all data as json. This function maps each element of the input bag to a json encoded string and writes one file per partition to the export_dir. WARNING: this function will delete any json files already present in export_dir.

Parameters

- **key_value_records** (Bag) – A dask bag containing dicts.
- **export_dir** (Path) – The location to write json files. Will erase any if present beforehand.

Return type None

agatha.util.test_proto_util module

agatha.util.test_proto_util.**test_load_json_as_proto**()

agatha.util.test_proto_util.**test_load_serialized_pb_as_proto**()

agatha.util.test_proto_util.**test_load_text_as_proto**()

agatha.util.test_proto_util.**test_parse_proto_fields_build_config**()

agatha.util.test_proto_util.**test_parse_proto_fields_ftp_source**()

agatha.util.test_proto_util.**test_set_field_nested**()

agatha.util.test_proto_util.**test_set_field_unnested**()

agatha.util.test_proto_util.**test_setup_parser_with_proto**()

agatha.util.test_proto_util.**test_transfer_args_to_proto**()

agatha.util.test_sqlite3_lookup module

agatha.util.test_sqlite3_lookup.**make_sqlite3_db**(*test_name*, *data*, *table_name='lookup_table'*, *key_column_name='key'*, *value_column_name='value'*, *tmp_dir=PosixPath('/tmp')*)

Return type Path

agatha.util.test_sqlite3_lookup.**test_backward_compatable_fallback**()

agatha.util.test_sqlite3_lookup.**test_custom_key_column_name**()

agatha.util.test_sqlite3_lookup.**test_custom_table_name**()

agatha.util.test_sqlite3_lookup.**test_custom_value_column_name**()

agatha.util.test_sqlite3_lookup.**test_iter**()

agatha.util.test_sqlite3_lookup.**test_iter_where**()

agatha.util.test_sqlite3_lookup.**test_keys**()

agatha.util.test_sqlite3_lookup.**test_len**()

agatha.util.test_sqlite3_lookup.**test_make_sqlite3_db**()

agatha.util.test_sqlite3_lookup.**test_new_sqlite3bow**()

agatha.util.test_sqlite3_lookup.**test_new_sqlite3graph**()

agatha.util.test_sqlite3_lookup.**test_old_sqlite3bow**()

agatha.util.test_sqlite3_lookup.**test_old_sqlite3graph**()

agatha.util.test_sqlite3_lookup.**test_sqlite3_is_preloaded**()

agatha.util.test_sqlite3_lookup.**test_sqlite3_lookup_contains**()

agatha.util.test_sqlite3_lookup.**test_sqlite3_lookup_getitem**()

agatha.util.test_sqlite3_lookup.**test_sqlite3_lookup_pickle**()

agatha.util.test_sqlite3_lookup.**test_sqlite3_lookup_preload**()

agatha.util.test_umls_util module

agatha.util.test_umls_util.**TEST_MRCONSO_PATH = PosixPath('test_data/tiny_MRCONSO.RRF')**
tiny_MRCONSO

Only contains the top-1000 lines from 2020AA Only contains the following UMLS terms:

agatha.util.test_umls_util.**test_codes_with_minimum_edit_distance**()

agatha.util.test_umls_util.**test_create_umls_index**()

agatha.util.test_umls_util.**test_create_umls_index_filter**()

agatha.util.test_umls_util.**test_filter_atoms_code_subset**()

agatha.util.test_umls_util.**test_filter_atoms_language_eng**()

agatha.util.test_umls_util.**test_filter_atoms_suppress_content**()

agatha.util.test_umls_util.**test_find_codes**()

agatha.util.test_umls_util.**test_has_code**()

agatha.util.test_umls_util.**test_has_pref_text**()

agatha.util.test_umls_util.**test_parse_first_line**()

agatha.util.test_umls_util.**test_parse_mrconso**()
Need to parse all 1000 lines of TEST_MRCONSO_PATH

agatha.util.umls_util module

umls_util.py

This module is responsible for cross referencing UMLS MRCONSO. This means that we will be able to both lookup UMLS terms from plaintext descriptions, and vice-versa.

class agatha.util.umls_util.**UmlsIndex**(*mrconso_path*, ***filter_kwargs*)
Bases: object

The UmlsIndex is responsible for managing the MRCONSO file.

When we create the UmlsIndex we create the intermediate data structures required to index all UMLS keywords, and all plaintext atoms. You can download a MRCONSO file associated with a UMLS release here:

www.nlm.nih.gov/research/umls/licensedcontent/umlsknowledgesources.html

Take a look to see what the MRCONSO file format is supposed to look like:

https://www.ncbi.nlm.nih.gov/books/NBK9685/table/ch03.T.concept_names_and_sources_file_mr/

Parameters

- **mrconso_path** (Path) – The path to a MRCONSO RRF file.
- **include_supressed_content** – By default, this index will only consider terms that have not been marked as *SUPPRESS*. If this flag is set, we will include all terms.
- **filter_language** – If set, this index will only consider names appearing in the selected langauge (default = ENG). If set to *None*, all terms will be considered.

codes()

Return type Set[str]

contains_code(*code*)

Return type bool

contains_pref_text_for_code(*code*)

Return type bool

find_codes_with_close_text(*text*, *ignore_case=False*)
Returns the set of codes with text most similar to that provided.

Each text field of all managed atoms is compared to the given text. The set of codes with text that minimize edit distance with the given text are returned.

For example, if codes C1 and C2 are both equally distant to text, then both will be returned.

Return type Set[str]

find_codes_with_pattern(*pattern*)
Returns the set of codes with text that matches the regex pattern

Return type Set[str]

get_pref_text(*code*)

Return type str

get_texts(*code*)

Return type Set[str]

num_codes()

Return type int

agatha.util.umls_util.**atom_contains_all_fields**(*atom*)

Return type bool

agatha.util.umls_util.**filter_atoms**(*mrconso_data*, *include_suppressed=False*, *filter_language='ENG'*, *code_subset=None*)

Filters the lines of MRCONSO

If *include_suppressed* is set, then atoms with *SUPPRESS* set will be included in the result.

If *filter_language* is not *None*, then only atoms with *LAT* set to the filter language will be included.

If *code_subset* is set, then only UMLS terms present in this set will be passed through the filter.

Return type Iterable[Dict[str, str]]

agatha.util.umls_util.**parse_mrconso**(*mrconso_path*)

Parses MRCONSO file

The MRCONSO file, as described in:

https://www.ncbi.nlm.nih.gov/books/NBK9685/table/ch03.T.concept_names_and_sources_file_mr/

Has columns described in *umls_util.MRCONSO_FIELDNAMES*.

This function takes each line of the MRCONSO.RRF file name parses out each field. The result is a list of dictionaries, where *parse_mrconso(. . .)[i]* contains all of the fields of line *i*. For instance, you can get the CUID of line *i* by calling *parse_mrconso(. . .)[i]['cui']*

Parameters **mrconso_path** (Path) – The filepath to MRCONSO.RRF. Must end in *.RRF*.

Return type Iterable[Dict[str, str]]

Returns List of parsed MRCONSO data. Each line contains the fields defined in MRCONSO_FIELDNAMES.

Module contents

2.1.2 Module contents

CHAPTER

THREE

HELP

3.1 How to Embed the Agatha Semantic Graph

We use Pytorch Big Graph (PTBG) to embed our semantic graph. This is a distributed knowledge graph embedding tool, meaning that it uses multiple machines, and takes node / edge type into account when embedding. PTBG is a complex tool that requires a number of preprocessing steps to use.

3.1.1 PTBG Process Outline

1. Create a single directory that contains all semantic graph edges.
    - This is produced by running `agatha.construct`.
    - Edges are stored as small key-value json files.
    - The directory may contain a large number of files.
2. Convert graph to PTBG input format.
    - PTBG requires that we index and partition all nodes and edges.
    - Look into `tools/convert_graph_for_pytorch_biggraph` for how to do this.
3. Create a PTBG config.
    - Specify all node / edge types.
    - Specify location of all input files.
    - The parameters of this config must match the options used in the conversion.
4. Launch the PTBG training cluster.
    - Use 10-20 machines, too many will slow this process.
    - Wait at least 5 epochs, will take days.
5. Index the resulting embeddings for use in Agatha.
    - Agatha needs to know where to find each embedding, given the node name.
    - Use `tools/py_scripts/ptbg_index_embeddings.py` to create a lookup table that maps each node name to its embedding metadata.

3.1.2 Convert Edges to PTBG format

The PTBG conversion tool is a multi-threaded single-machine program that indexes every node and edge of the input graph for PTBG distributed training. The settings used to run this tool will determine qualities of the resulting PTBG config, so you will want to save the exact command you run for later steps.

Warning: This program is extremely memory intensive. If you're running on plametto, make sure to grab the 1.5 or 2 TB node.

To begin, build the `convert_graph_for_pytorch_biggraph` tool.

```
cd /path/to/agatha/tools/convert_graph_for_pytorch_biggraph
make
```

This will produce `graph_to_ptbg`. You can take a look at how this tool works with the `./graph_to_ptbg --help` command.

If you want to embed the entire graph, you can run this conversion with:

```
./graph_to_ptbg -i <json_edge_dir> -o <ptbg_data_dir>
```

By default, this will include all expected node and relationship types, as described in the Agatha paper.

If you only want to embed part of the graph, you can select the specific node and relation types to include. Note that excluded types will be ignored.

To select as subset of nodes, you will need to supply the optional `--types` and `--relations` arguments. Here's an example of using these flags to select only nodes and relationships between umls terms (type m) and predicates (type p).

```
./graph_to_ptbg \
  -i <json_edge_dir> \
  -o <ptbg_data_dir> \
  --types "mp" \
  --relations "mp pm"
```

Note that the argument passed with `--types` should be a string where each character indicates a desired node type. Nodes of types outside of this list will not be included in the output.

Note that the argument passed with `--relations` should be a string with space-separated relationship types. Each relationship should be a two character long string. Relationships are also directed in PTBG, meaning that if you would like to select both `UMLS -> predicate` edges, as well as `predicate -> UMLS` edges, you will need to specify both edge types.

3.1.3 Create a PTBG Config

Now that you have converted the agatha semantic graph for PTBG, you now need to write a configuration script. Here's the official docs for the PTBG config. The following is an example PTBG config. The parts you need to worry about occur in the header section of the `get_torchbiggraph_config` function. You should copy this and change what you need.

```
#!/usr/bin/env python3
def get_torchbiggraph_config():

    # CHANGE THESE ##########################################################

    DATA_ROOT = "/path/to/data/root"
```

(continues on next page)

(continued from previous page)

```
    """ This is the location you specified with the `-o` flag when running
    `convert_graph_for_pytorch_biggraph` That tools should have created
    `DATA_ROOT/entities` and `DATA_ROOT/edges`. This process will create
    `DATA_ROOT/embeddings`.  """

    PARTS = 100
    """ This is the number of partitions that all nodes and edges have been
    split between when running `convert_graph_for_pytorch_biggraph`. By default,
    we create 100 partitions. If you specified `--partition-count` (`-c`), then
    you need to change this value to reflect the new partition count.  """

    ENT_TYPES = "selmnp"
    """ This is the set of entities specified when running
    `convert_graph_for_pytorch_biggraph`. The above value is the default. If you
    used the `--types` flag, then you need to set this value accordingly."""

    RELATIONS = [ "ep", "es", "lp", "ls", "mp", "ms", "np", "ns", "pe", "pl",
                  "pm", "pn", "ps", "se", "sl", "sm", "sn", "sp", "ss" ]
    """ This is the ordered list of relationships that you specified when
    running `convert_graph_for_pytorch_biggraph`. The above is the default. If
    you specified `--relations` then you need to set this value accordingly.
    """

    EMBEDDING_DIM = 512
    """ This is the number of floats per embedding per node in the resulting
    embedding. """

    NUM_COMPUTE_NODES = 20
    """ This is the number of computers used to compute the embedding. We find
    that around 20 machines is the sweet spot. More or less result in slower
    embeddings. """

    THREADS_PER_NODE = 24
    """ This is the number of threads that each machine will use to compute
    embeddings. """

    ######################################################################

    config = dict(
        # IO Paths
        entity_path=DATA_ROOT+"/entities",
        edge_paths=[DATA_ROOT+"/edges"],
        checkpoint_path=DATA_ROOT+"/embeddings",

        # Graph structure
        entities={t: {'num_partitions': PARTS} for t in ENT_TYPES},
        relations=[
          dict(name=rel, lhs=rel[0], rhs=rel[1], operator='translation')
          for rel in sorted(RELATIONS)
        ],

        # Scoring model
        dimension=EMBEDDING_DIM,
        comparator='dot',
        bias=True,

        # Training
```

(continues on next page)

(continued from previous page)

```
        num_epochs=5,
        num_uniform_negs=50,
        loss_fn='softmax',
        lr=0.02,

        # Evaluation during training
        eval_fraction=0,

        # One per allowed thread
        workers=THREADS_PER_NODE,
        num_machines=NUM_COMPUTE_NODES,
        distributed_init_method="env://",
        num_partition_servers=-1,
    )

    return config
```

3.1.4 Launch the PTBG training cluster

Now there is only a little more book keeping nessesary to launch PTBG distributed training. This next step will look familiar to you if you've already taken a look at the docs for training the agatha deep learning model. Afterall, both techniques are using pytorch distributed.

You need every machine in your compute cluster to have some environment variables set. The offical docs on pytorch distributed environemnt variables can be found here.

These varaibles are:

`MASTER_ADDR` : The hostname of the master node. In our case, this is just one of the workers.

`MASTER_PORT` : An unused port to communicate. This can almost be anything. We use `12910`.

`NODE_RANK` : The rank of this machine with respect to the "world". If there are 3 total machines, then each should have `NODE_RANK` set to a different value in: [0, 1, 2]. Note, we use `NODE_RANK` but default pytorch uses `RANK`. It doesn't really matter as long as you're consistent.

`WORLD_SIZE` : The total number of machines.

The simplest way to set these variables is to use a `nodefile`, which is just a file that contains each machine's address. If you're on PBS, you will have a file called `$PBS_NODEFILE` and if you're on SLURM then you will have variable called `$SLURM_NODELIST`. To remain platform agnositic, we assume you have copied any nessesry nodefile to `~/.nodefile`.

You can set all of the nessesary variables with this snippet, run on each node. Preferably, this should be somewhere in your `~.bashrc`.

```
export NODEFILE="~/.nodefile"
export NODE_RANK=$(grep -n $HOSTNAME $NODEFILE | awk 'BEGIN{FS=":"}{print $1-1}')
export MASTER_ADDR=$(head -1 $NODEFILE)
export MASTER_PORT=12910
export WORLD_SIZE=$(cat $NODEFILE | wc -l)
```

Now that you've setup the appropriate environment variables, you're ready to start training. To launch a training process for each compute node, we're going to use `parallel`.

```
parallel \
  --sshloginfile $NODEFILE \
```

(continues on next page)

(continued from previous page)

```
  --ungroup \
  --nonall \
  "torchbiggraph_train --rank \$NODE_RANK /path/to/config.py"
```

Warning: It is important to escape the '$' when calling `$NODE_RANK` in the above parallel command. With the escape (`\$`) we're saying "evaluate `NODE_RANK` on the remote machine". Without the escape we're saying "evaluate `NODE_RANK` on *this* machine."

Expect graph embedding to take a very long time. Typically, we run for approximately 5 epochs, which is the most we can compute in our normal 72-hour time limit.

3.1.5 Index the Resulting Embeddings

Now, you should have a directory in `DATA_ROOT/embeddings` that is full of files that look like: `embeddings_{type}_{part}.v{epoch}.h5`. Each one represents a matrix where row `i` corresponds to entity `i` of the given type and partition. Therefore, we need to build a small index that helps us reference each embedding given an entity name. For this, we use the `ptbg_index_embeddings.py` tool. You can find this here: `agatha/tools/py_scripts/ptbg_index_embeddings.py`.

Like all tools, you can use the `--help` option to get more information. Here's an example of how to run it:

```
cd /path/to/agatha/tools/py_scripts
./ptbg_index_embeddings.py \
  <DATA_ROOT>/entities \
  entities.sqlite3
```

This tool will create an sqlite3 lookup table (compatible with `Sqlite3LookupTable`) that maps each entity to its embedding location. The result will be stored in `entities.sqlite3` (or wherever you specify). The Agatha embedding lookup table will use this along with the `embeddings` directory to rapidly lookup vectors for each entity.

3.2 Pretrained Models

3.2.1 AGATHA 2015

DOWNLOAD AGATHA 2015

The 2015 version of the Agatha model was trained on 2020-05-13. This model uses all Medline abstracts dating before 2015-01-01. This model is used to validate the performance of Agatha. Use this model to replicate our experiments from the Agatha publication. Note, this model is a retrained version of the same model used to report numbers.

Contents

```
model_release/
  model.pt
  predicate_embeddings/
    embeddings_*.v5.h5
  predicate_entities.sqlite3
  predicate_graph.sqlite3
```

Model Training Parameters

```
{
  'logger':                          True,
  'checkpoint_callback':             True,
  'early_stop_callback':             False,
  'gradient_clip_val':               1.0,
  'process_position':                0,
  'num_nodes':                       5,
  'num_processes':                   1,
  'gpus':                            '0,1',
  'auto_select_gpus':                False,
  'num_tpu_cores':                   None,
  'log_gpu_memory':                  None,
  'progress_bar_refresh_rate':       1,
  'overfit_pct':                     0.0,
  'track_grad_norm':                 -1,
  'check_val_every_n_epoch':         1,
  'fast_dev_run':                    False,
  'accumulate_grad_batches':         1,
  'max_epochs':                      10,
  'min_epochs':                      1,
  'max_steps':                       None,
  'min_steps':                       None,
  'train_percent_check':             0.1,
  'val_percent_check':               0.1,
  'test_percent_check':              1.0,
  'val_check_interval':              1.0,
  'log_save_interval':               100,
  'row_log_interval':                10,
  'distributed_backend':             'ddp',
  'precision':                       16,
  'print_nan_grads':                 False,
  'weights_summary':                 'full',
  'num_sanity_val_steps':            3,
  'truncated_bptt_steps':            None,
  'resume_from_checkpoint':          None,
  'benchmark':                       False,
  'reload_dataloaders_every_epoch': False,
  'auto_lr_find':                    False,
  'replace_sampler_ddp':             True,
  'progress_bar_callback':           True,
  'amp_level':                       'O1',
  'terminate_on_nan':                False,
  'dataloader_workers':              3,
  'dim':                             512,
  'lr':                              0.02,
  'margin':                          0.1,
  'negative_scramble_rate':          10,
  'negative_swap_rate':              30,
  'neighbor_sample_rate':            15,
  'positives_per_batch':             80,
  'transformer_dropout':             0.1,
  'transformer_ff_dim':              1024,
  'transformer_heads':               16,
  'transformer_layers':              4,
  'validation_fraction':             0.2,
```

(continues on next page)

(continued from previous page)

```
  'verbose':                          True,
  'warmup_steps':                     100,
  'weight_decay':                     0.01,
  'disable_cache':                    False
}
```

3.2.2 AGATHA 2020

DOWNLOAD AGATHA 2020

The 2020 version of the Agatha model was trained on 2020-05-04. This model uses all available Medline abstracts as well as all available predicates in the most up-to-date release of SemMedDBsemmeddb. *This model does NOT contain any COVID-19 related terms or customizations.*

Contents

```
model_release/
  model.pt
  predicate_embeddings/
    embeddings_*.v5.h5
  predicate_entities.sqlite3
  predicate_graph.sqlite3
```

Data Construction Parameters

```
cluster {
  address: "10.128.3.160"
  shared_scratch: "/scratch4/jsybran/agatha_2020"
  local_scratch: "/tmp/agatha_local_scratch"
}
parser {
  # This is the code for scibert in huggingface
  bert_model: "monologg/scibert_scivocab_uncased"
  scispacy_version: "en_core_sci_lg"
  stopword_list: "/zfs/safrolab/users/jsybran/agatha/data/stopwords/stopword_list.txt"
}
sentence_knn {
  num_neighbors: 25
  training_probability: 0.005
}
sys {
  disable_gpu: true
}
phrases {
  min_ngram_support_per_partition: 10
  min_ngram_support: 50
  ngram_sample_rate: 0.2
}
```

Model Training Parameters

```
{
  'accumulate_grad_batches':          1
  'amp_level':                        'O1'
  'auto_lr_find':                     False
  'auto_select_gpus':                 False
  'benchmark':                        False
  'check_val_every_n_epoch':          1
  'checkpoint_callback':              True
  'dataloader_workers':               3
  'dim':                              512
  'distributed_backend':              'ddp'
  'early_stop_callback':              False
  'fast_dev_run':                     False
  'gpus':                             '0,1'
  'gradient_clip_val':                1.0
  'log_gpu_memory':                   None
  'log_save_interval':                100
  'logger':                           True
  'lr':                               0.02
  'margin':                           0.1
  'max_epochs':                       10
  'max_steps':                        None
  'min_epochs':                       1
  'min_steps':                        None
  'negative_scramble_rate':           10
  'negative_swap_rate':               30
  'neighbor_sample_rate':             15
  'num_nodes':                        10
  'num_processes':                    1
  'num_sanity_val_steps':             3
  'num_tpu_cores':                    None
  'overfit_pct':                      0.0
  'positives_per_batch':              80
  'precision':                        16
  'print_nan_grads':                  False
  'process_position':                 0
  'progress_bar_callback':            True
  'progress_bar_refresh_rate':        1
  'reload_dataloaders_every_epoch':   False
  'replace_sampler_ddp':              True
  'resume_from_checkpoint':           None
  'row_log_interval':                 10
  'terminate_on_nan':                 False
  'test_percent_check':               1.0
  'track_grad_norm':                  -1
  'train_percent_check':              0.1
  'transformer_dropout':              0.1
  'transformer_ff_dim':               1024
  'transformer_heads':                16
  'transformer_layers':               4
  'truncated_bptt_steps':             None
  'val_check_interval':               1.0
  'val_percent_check':                0.1
  'validation_fraction':              0.2
  'verbose':                          True
```

(continues on next page)

(continued from previous page)

```
  'warmup_steps':                    100
  'weight_decay':                    0.01
  'weights_summary':                 'full'
}
```

3.3 How to Run Your Agatha Model

The first set to run your agatha model is to either train your own, or download a pretrained model. Other help pages describe how to do this in more detail.

To begin, lets assume you downloaded Agatha 2020. When extracted, you should see the following directory structure. This is consistent with other pretrained models.

```
model_release/
  model.pt
  predicate_embeddings/
    embeddings_*.v5.h5
  predicate_entities.sqlite3
  predicate_graph.sqlite3
```

3.3.1 TL;DR

Once you have the necessary files, you can run the model with the following snippet:

```
# Example paths, change according to your download location
model_path = "model_release/model.pt"
embedding_dir = "model_release/predicate_embeddings"
entity_db_path = "model_release/predicate_entities.sqlite3"
graph_db_path = "model_release/predicate_graph.sqlite3"

# Load the model
import torch
model = torch.load(model_path)

# Configure auxilary data paths
model.configure_paths(
  embedding_dir=embedding_dir
  entity_db=entity_db_path,
  graph_db=graph_db_path,
)

# Now you're ready to run some predictions!
# C0006826 : Cancer
# C0040329 : Tobacco
model.predict_from_terms([("C0006826", "C0040329")])
>>> [0.9946276545524597]

# Speed up by moving the model to GPU
model = model.cuda()

# If we would like to run thousands of queries, we want to load everything
# before the query process. This takes a while, and is optional.
```

(continues on next page)

(continued from previous page)

```
model.preload()

# Get a list of valid terms (slow if preload not called beforehand)
from agatha.util.entity_types import is_umls_term_type
valid_terms = list(filter(is_umls_term_type, model.graph.keys()))
len(valid_terms)
>>> 278139

# Run a big batch of queries
from random import choice
rand_term_pairs =  [
 (choice(valid_terms), choice(valid_terms))
 for _ in range(100)
]
scores = model.predict_from_terms(rand_term_pairs, batch_size=64)
# Now, scores[i] corresponds to rand_term_pairs[i]
```

3.3.2 Required Files

`model.pt` : This is the model file. You can load this with `torch.load`. Behind the scenes, this will startup the appropriate Agatha module.

`predicate_embeddings` : This directory contains graph embeddings for each entity needed to make predicate predictions using the Agatha model.

`predicate_entities.sqlite3` : This database contains embedding metadata for each entity managed by the Agatha model. This database is loaded with `agatha.util.sqlite3_lookup.Sqlite3LookupTable`.

`predicate_graph.sqlite3` : This database contains term-predicate relationships for each entity managed by the Agatha model. This database is loaded with `agatha.util.sqlite3_lookup.Sqlite3LookupTable`.

3.3.3 Bulk queries

In order to run bulk queries efficiently, you will want to run:

```
model.cuda()
model.preload()
```

The first command, `model.cuda()` moves the weight matrices to the GPU. The second command, `model.preload()` modes all graph and embedding information into RAM. This way, each request for an embedding, of which we will do tens of times per query, can be handled without a slow lookup in the storage system. Warning, expect this to take around 30 GB of RAM to start. Additionally, Agatha employs caching intermediate values that will increase the memory usage as the query process goes on.

Batch size

When running `model.predict_from_terms` the optional `batch_size` parameter can be used to improve GPU usage. Set this value to an integer greater than one to pack more than one query within each call to the GPU. You may need to experiment to find a value that is large, but doesn't exceed GPU memory.

3.4 Topic Model Queries on the Agatha Semantic Network

Our prior work, Moliere, performed hypothesis generation through a graph-analytic and topic-modeling approach. Occasionally, we would like to run this same approach using the Agatha topic network. This document describes the way to use the `agatha.topic_query` module to perform topic-model queries, and how to interpret your results.

3.4.1 TL;DR

This is the recommended way to run the query process. First, create a file called `query.conf` and fill it with the following information:

```
graph_db: "<path to graph.sqlite3>"
bow_db: "<path to sentences.sqlite3>"
topic_model {
  num_topics: 100
}
```

Look into `agatha/topic_query/topic_query_config.proto` to get more details on the `TopicQueryConfig` specification.

Now you can run queries using the following syntax:

```
python3 -m agatha.topic_query query.conf \
  --source <source term> \
  --target <target term> \
  --result_path <desired place to put result>
```

Here is a real-life example of a query:

```
python3 -m agatha.topic_query configs/query_2020.conf \
  --source l:noun:tobacco \
  --target l:noun:cancer \
  --result_path ./tobacco_cancer.pb
```

3.4.2 Viewing Results

Once you're done your query, you will have a binary file containing all topic model information. This is stored as a compressed proto format, which should enable easy programmatic access to all the components of the query result. You can view more details on the proto specification at `agatha/query/topic_query_result.proto`.

Here's a short python script that would load a proto result file for use:

```
from agatha.topic_query import topic_query_result_pb2
result = topic_query_result_pb2.TopicQueryResult()
with open("<result path>", 'rb') as proto_file:
  result.ParseFromString(proto_file.read())
```

You now have access to: `result.path`, `result.documents`, and `result.topics`.

If you want to cut to the chase, you can simply print out all proto result details using the following script:

3.4.3 Running Queries with Node Names

In order to run queries, you will need to know the particular node names of the elements you would like to explore. Nodes of the Agatha network can be explored by looking at the set of `node` entities in the graph database. You can explore these in `sqlite3` with the following syntax:

```
sqlite3 .../graph.sqlite3 \
  'select node from graph where node like "%<query term>%" limit 10'
```

Here's an actual example:

```
sqlite3 graph.sqlite3 'select node from graph where node like "%dimentia%" limit 10'
  > e:amyotrophic_lateral_sclerosis/parkinsonism_dimentia_complex
  > e:dimentia_complex
  > e:hiv-associated_dimentia
  > e:mild_dimentia
  > e:three-dimentianl_(_3d_)
  > l:adj:three-dimentianl
  > l:noun:dimentia
```

Note that node names follow particular patterns. All valid node names start with a leading "type" character. These are specified in `agatha/util/entity_types.py`. Here are the existing entity types at the time of writing:

```
ENTITY_TYPE="e"
EMMA_TYPE="l"
MESH_TERM_TYPE="m"
UMLS_TERM_TYPE="m"
NGRAM_TYPE="n"
PREDICATE_TYPE="p"
SENTENCE_TYPE="s"
```

3.4.4 Configuration

Just like the Agatha network construction process, the query process also needs many parameters that are specified either through command-line arguments, or through a configuration script. We recommend creating a configuration for the typical query case, omitting only the query term parameters. This way you can have the simplest query interface when running these topic-model queries yourself.

Look into `agatha/config/topic_query_config.proto` to get more details on the `TopicQueryConfig` specification. Here is a fuller example of a configuration that we actually use on Palmetto.

```
# TopicQueryConfig

# source: Omitted
# target: Omitted
# result_path: Omitted

graph_db: "/zfs/safrolab/users/jsybran/agatha/data/releases/2020/graph.sqlite3"
bow_db: "/zfs/safrolab/users/jsybran/agatha/data/releases/2020/sentences.sqlite3"
topic_model {
  num_topics: 20
```

(continues on next page)

(continued from previous page)

```
  min_support_count: 2
  truncate_size: 250
}

# Advanced

max_sentences_per_path_elem: 2000
max_degree: 1000
```

3.5 How to Train Agatha

Training the Agatha deep learning model is the last step to generating hypotheses after you've already processed all necessary information using `agatha.construct`. This process uses PyTorch and PyTorch-Lightning to efficiently manage the distributed training of the predicate ranking model stored in `agatha.ml.hypothesis_predictor`.

3.5.1 tl:dr;

You will need the following files:

- `predicate_graph.sqlite3`
- `predicate_entities.sqlite3`
- `embeddings/predicate_subset/*.h5`

You will need to run `python3 -m agatha.ml.hypothesis_predictor` with the right keyword arguments. If performing distributed training, you will need to run this on each machine in your training cluster.

Take a look at [scripts/train_2020.sh][https://github.com/JSybrandt/agatha/blob/master/scripts/train_2020.sh] for how to train the agatha model.

If you are running the training process on one machine and only one gpu, you will want to remove the `distributed_backend` flag, and make sure `num_nodes` is set to one. If you are using multiple machines, or *multiple gpus on one machine*, then you will want to make sure that `distributed_backend="ddp"` and you should take a look at setting the distributed environment variables if you run into errors. In the multi-gpu one-machine case, these variables should be set automatically.

3.5.2 Background

The Agatha deep learning model learns to rank entity-pairs. To learn this ranking, we will be comparing existing predicates found within our dataset against randomly sampled entity-pairs. Of course, if a predicate exists in our database, it should receive a higher model output than many random pairs.

A `positive sample` is a entity-pair that actually occurs in our dataset. A `negative sample` is one of those non-existent randomly sampled pairs. We will use the margin ranking loss criteria to learn to associate higher values with positive samples. To do this, we will compare one positive sample to a high number of negative samples. This is the `negative-sampling rate`.

A single sample, be it positive or negative, is comprised of four parts:

1. Term 1 (the subject).
2. Term 2 (the object).
3. Predicates associated with term 1 (but not term 2).

4. Predicates associated with term 2 (but not term 1).

This as a whole is referred to as a `sample`. Generating samples is the primary bottleneck in the training process. This is because we have many millions of terms and predicates. As a result, the Agatha deep learning framework comes along with a number of utilities to make managing the large datasets easier.

3.5.3 Datasets

In order to begin training you will need the following data:

1. Embeddings for all entities and predicates, stored as a directory of `.h5` files.
2. Entity metadata, stored as a `.sqlite3` file.
3. The subgraph containing all entity-predicate edges, stored as a `.sqlite3` file.

The network construction process will produce these datasets as `sqlite3` files. Sqlite is an embedded database, meaning that we can load the database from storage and don't need to spin up a whole server. Additionally, because we are only going to *read* and never going to *write* to these databases during training, each machine in our distributed training cluster can have independent access to the same data very efficiently.

All of the sqlite3 databases managed by Agatha are stored in a simple format that enables easy python access through the `agatha.util.sqlite3_lookup.Sqlite3LookupTable` object. This provides read-only access to the database through a dictionary-like interface.

For instance, if we want to get the neighbors for the node `l:noun:cancer`, we can simply write this code:

```
from agatha.util.sqlite3_lookup import Sqlite3LookupTable
graph = Sqlite3LookupTable("./data./releases/2020/graph.sqlite3")
graph["l:noun:cancer"]
# Returns:
# ... [
# ...   < List of all neighbors >
# ... ]
```

This process works by first making an sqlite3 connection to the graph database file. By default, we assume that this database contains a table called `lookup_table` that has the schema: `(key:str, value:str)`. Values in this database are all json-encoded. This means that calling `graph[foo]` looks up the value associated with "foo" in the database, and parses whatever it find through `json.loads(...)`.

This process is slow compared to most other operations in the training pipeline. Each query has to check against the sqlite `key` index, which is stored on disk, load the `value`, also stored on disk, and then parse the string. There are two optimizations that make this faster: preloading and caching. Look into the API documentation for more detail.

3.5.4 Installing Apex for AMP

Apex is a bit of a weird dependency, but it allows us to take advantage of some GPU optimizations that really cut back our memory footprint. Amp allows us to train using 16-bit precision, enabling more samples per batch, resulting in faster training times. However, note that if you install apex on a node that has one type of GPU, you will get an error if you try and train on another. This means that you **need** to install this dependency on a training node with the appropriate GPU.

Warning: Apex is going to require a different version of GCC than we typically use. If you're on palmetto, you can run: `module rm gcc/8.1.0; module load gcc/6.3.0`

To install apex, first select a location such as `~/software` to keep the files. Next, download apex with git `git clone https://github.com/NVIDIA/apex.git`. Finally, install the dependency with: `pip install -v --no-cache-dir --global-option="--cpp_ext" --global-option="--cuda_ext" ./`

In full, run this:

```
# SSH into one of your training nodes with the correct GPU configuration.
# Make sure the appropriate modules are loaded.
# Assuming you would like to install apex in ~/software/apex

# make software dir if its not present
mkdir -p ~/software

# Clone apex to ~/software/apex
git clone https://github.com/NVIDIA/apex.git ~/software/apex

# Enter Apex dir
cd ~/software/apex

# Run install
pip install -v                       \
  --no-cache-dir                     \
  --global-option="--cpp_ext"        \
  --global-option="--cuda_ext"       \
  ./
```

3.5.5 Model Parameters

This is *NOT* an exhaustive list of the parameters present in the Agatha deep learning model, but is a full list of the parameters you need to know to train the model.

`amp_level` : The optimization level used by NVIDIA. `O1` works well. `O2` causes some convergence issues, so I would stay away from that.

`default_root_dir` : The directory to store model training files.

`dataloader-workers` : The number of processes used to generate predicate pairs, per-gpu. Too many dataloader workers will cause an out-of-memory error. I've found 3 works well.

`dim` : The number of dimensions of each input embedding. We use 512 in most cases. This parameter effects the size of various internal parameters.

`distributed_backend` : Used to specify how to communicate between GPUs. Ignored if using only one GPU. Set to `ddp` for distributed data parallel (even if only using gpus on the same node).

`embedding-dir` : The system path containing embedding `HDF5` (`*.h5`) files.

`entity-db` : The system path to the entities `.sqlite3` database.

`gpus` : The specific GPUs enabled on this machine. GPUs are indexed starting from 0. On a 2-GPU node, this should be set to `0,1`.

`gradient_clip_val` : A single step of gradient decent cannot move a parameter more than this amount. We find that setting this to `1.0` enables convergence.

`graph-db` : The system path to the graph `.sqlite3` database.

`lr` : The learning rate. We use `0.02` because we're cool.

`margin` : The objective of the Agatha training procedure is Margin Ranking Loss. This parameter determines how different a positive ranking criteria needs to be from all negative ranking criteria. Setting this too high or low will cause convergence issues. Remember that the model outputs in the `[0,1]` interval. We recommend `0.1`.

`max_epochs` : The maximum number of times to go through the training set.

`negative-scramble-rate` : For each positive sample, how many negative scrambles (easy negative samples).

`negative-swap-rate` : For each positive sample, how many negative swaps (hard negative samples).

`neighbor-sample-rate` : When sampling a term-pair, we also sample each pair's disjoint neighborhood. This determines the maximum number of neighbors to include.

`num_nodes` : This determines the number of *MACHINES* used to train the model.

`num_sanity_val_steps` : Before starting training in earnest, we can optionally take a few validation steps just to make sure everything has been configured properly. If this is set above zero, we will run multiple validation steps on the newly instantiated model. Recommended to run around `3` just to make sure everything is working.

`positives-per-batch` : Number of positive samples per batch per machine. More results in faster training. Keep in mind that the true batch size will be `num_nodes * positives-per-batch * (negative-scramble-rate + negative-swap-rate)`. When running with 16-bit precision on V100 gpus, we can handle around `80` positives per batch.

`precision` : The number of bits per-float. Set to `16` for half-precision if you've installed apex.

`train_percent_check` : Limits the number of actual training examples per-epoch. If set to `0.1` then one epoch will occur after every 10% of the training data. This is important because we only checkpoint after every epoch, and don't want to spend too much time computing between checkpoints. We recommend that if you set this value, you should increase `max_epochs` accordingly.

`transformer-dropout` : Within the transformer encoder of Agatha, there is a dropout parameter that helps improve performance. Recommended you set this to `0.1`.

`transformer-ff-dim` : The size fo the transformer-encoded feed-forward layer. Recommended you set this to something between `2*dim` and `4*dim`.

`transformer-heads` : The number of self-attention operations per self-attention block in the transformer encoder. We use `16`.

`transformer-layers` : The number of transformer encoder blocks. Each transformer-encoder contains multi-headed self-attention and a feed-forward layer. More transformer encoder layers should lead to higher quality, but will require additional training time and memory.

`val_percent_check` : Just like how `train_percent_check` limits the number of training samples per-epoch, `val_percent_check` limits the number of validation samples per-epoch. Recommended that if you set one, you set the other accordingly.

`validation-fraction` : Before training, this parameter determines the training-validation split. A higher value means less training data, but more consistent validation numbers. Recommended you set to `0.2`.

`warmup-steps` : Agatha uses a gradient warmup strategy to improve early convergence. This parameter indicates the number of steps needed to reach the input learning rate. For instance, if you specify a learning rate of `0.02` and `100` warmup steps, at step `50` there will be an effective learning rate around `0.01`. We set this to `100`, but higher can be better if you have the time.

`weight-decay` : Each step, the weights of the agatha model will be moved towards zero at this rate. This helps with latter convergence and encourages sparsity. We set to `0.01`.

`weights_save_path` : The result root directory. Model checkpoints will be stored in `weights_save_path/checkpoints/version_X/`. Recommended that this is set to the same value as `default_root_dir`.

3.5.6 Subset Data with Percent Check Flags

In the list of model flags are two that deserve more explanation: `train_percent_check`, and `val_percent_check`. When debugging the model training process to ensure everything has been setup correctly, it is worthwhile to run the training routine through a couple of epochs quickly. This will ensure that the model output checkpoints are created properly. To do so, set `train_percent_check` and `val_percent_check` to a very small value, such as `0.0001`. Preferably, this will be small enough to complete an epoch in a couple of minutes. Warning, you set this value too low, you will filter out *all* of the training data and will create problems.

When you *actually* want to train the model, you still might want a modest `train_percent_check` and `val_percent_check`. For instance, if the estimated time per epoch is greater than a couple of hours, you might want more frequent check pointing. What we want to avoid is the amount of training time that is lost when an unpredictable system failure causes an outage 40 hours into training, and we haven't created our first checkpoint yet. If this were to happen, we would simply lose all of the progress we had made for nearly two days worth of computational effort.

Therefore, I recommend setting these values to something that reduces the time per epoch to the single-digit hours. Keep in mind that when you reduce the training set, and especially when you reduce the validation set, you should expect poorer convergence in the final model. Therefore, if at all possible, it is recommend that you increase the number of training processes by adding more distributed workers. Once you have as many machines as you can afford, then tune this parameter.

3.5.7 Running Distributed Training

In order to preform distributed training, you will need to ensure that your training cluster is each configured with the same modules, libraries, and python versions.

On palmetto, and many HPC systems, this can be done with modules and Anaconda. I recommend adding a section to your `.bashrc` for the sake of training Agatha that loads all necessary modules and activates the appropriate conda environment. As part of this configuration, you will need to set some environment variables on each machine that help coordinate training. These are `MASER_ADDR`, `MASTER_PORT`, and `NODE_RANK`.

Distributed Training Env Variables

`MASER_ADDR` : Needs to be set to the hostname of one of your training nodes. This node will coordinate the others.

`MASTER_PORT` : Needs to be set to an unused network port for each machine. Can be any large number. We recommend: `12910`.

`NODE_RANK` : If you have N machines, then each machine needs a unique `NODE_RANK` value between 0 and N-1.

We recommend setting these values automatically using a `nodefile`. A `nodefile` is just a text file containing the hostnames of each machine in your training cluster. The first name will be the `MASTER_ADDR` and the `NODE_RANK` will correspond to the order of names in the file.

If `~/.nodefile` is the path to your nodefile, then you can set these values with:

```
export NODEFILE=$HOME/.nodefile
export NODE_RANK=$(grep -n $HOSTNAME $NODEFILE | awk 'BEGIN{FS=":"}{print $1-1}')
export MASTER_ADDR=$(head -1 $NODEFILE)
export MASTER_PORT=12910
```

If you're on palmetto, you've already got access to the nodefile referenced by `PBS_NODEFILE`. However, only the first machine will have this variable set. I recommend automatically copying this file to some shared location whenever it is detected. You can do that in `.bashrc` by putting the following lines *BEFORE* setting the `NODE_RANK` and `MASER_ADDR` variables.

```
# If $PBS_NODEFILE is a file
if [[ -f $PBS_NODEFILE ]]; then
  cp $PBS_NODEFILE ~/.nodefile
fi
```

Launching Training on Each Machine with Parallel

Once each machine is configured, you will then need to run the agatha training module on each. We recommend `parallel` to help you do this. Parallel runs a given bash script multiple times simultaneously, and has some flags that let us run a script on each machine in a nodefile.

Put simply, you can start distributed training with the following:

```
parallel \
  --sshloginfile $NODEFILE \
  --ungroup \
  --nonall \
  python3 -m agatha.ml.hypothesis_predictor \
    ... agatha args ...
```

To explain the parameters:

`sshloginfile` : Specifies the set of machines to run training on. We use the `NODEFILE` created in the previous step.

`ungroup` : By default, `parallel` will wait until a process exits to show us its output. This flag gives us input every time a process writes the newline character.

`nonall` : This specifies that the following command (`python3`) will not need its arguments set by `parallel`, and that we would like to run the following command as-is, once per machine in `$NODEFILE`.

3.5.8 Palmetto-Specific Details

On palmetto, there are a number of modules that you will need to run Agatha. Here is what I load on every machine I use to train agatha:

```
# C++ compiler modules
module load gcc/8.3.0
module load mpc/0.8.1

# NVIDIA modules
module load cuda-toolkit/10.2.89
module load duDNN/10.2.v7.6.5
module load nccl/2.6.4-1

# Needed for parallel
module load gnu-parallel

# Needed to work with HDF5 files
module load hdf5/1.10.5

# Needed to work with sqlite
module load sqlite/3.21.0

conda activate agatha
```

(continues on next page)

(continued from previous page)

```
# Copy PBS_NODEFILE if it exists
if [[ -f $PBS_NODEFILE ]]; then
  cp $PBS_NODEFILE ~/.nodefile
fi

# Set distributed training variables
export NODEFILE="~/.nodefile"
export NODE_RANK=$(grep -n $HOSTNAME $NODEFILE | awk 'BEGIN{FS=":"}{print $1-1}')
export MASTER_ADDR=$(head -1 $NODEFILE)
export MASTER_PORT=12910
```

3.6 Loading the Trained Model

Once you've completed a few epochs of training, you will hopefully a see a file appear in `{weights_save_path}/lightning_logs/version_{#}/checkpoints/epoch={#}.ckpt`

If course, `weights_save_path` refers to whatever directory you listed in `--weights_save_path` in the training command-line arguments. The version number refers to the model version that pytorch-lightning deduces while training. Each time you run the training script with the same checkpoint directory, this number will increment. Then the epoch number will refer to whatever epoch this model last updated its checkpoint. Note here that the epoch number might be less than the number of epochs you've actually computed, because we will only update the checkpoint when the validation loss is improved.

To load the checkpoint in python, use:

```
from agatha.ml.hypothesis_predictor import HypothesisPredictor
model = HypothesisPredictor.load_from_checkpoint( ... )
```

When you want to give this model to someone else, you often don't want to give them the whole checkpoint. For this, you can use a simpler pytorch model format. The conversion is really simple:

```
checkpoint_path = ...
output_path = ...
import torch
from agatha.ml.hypothesis_predictor import HypothesisPredictor

# Load model from checkpoint
model = HypothesisPredictor.load_from_checkpoint(checkpoint_path)
# Save model in pytorch model format
torch.save(model, output_path)
```

The reason to do this is so future users can load your model with:

```
import torch
model = torch.load(...)
```

3.6.1 Running your new model.

Now that you have a model that you can load (either through `load_from_checkpoint` or `torch.load`, you can run some examples to ensure that everything has been configured properly. The simplest way to do this is to run a little script like this in your python terminal:

```
from agatha.ml.hypothesis_predictor import HypothesisPredictor
model = HypothesisPredictor.load_from_checkpoint("...")
# - OR -
import torch
model = torch.load("...")

# Configure auxilary data paths
model.configure_paths(
  embedding_dir="/path/to/embeddings",
  entity_db="/path/to/entities.sqlite3",
  graph_db="/path/to/graph.sqlite3",
)

# Optional, if you're going to do a lot of queries.
model = model.eval()
model.preload()

# C0006826 is the term for Tobacco
# C0040329 is the term for Cancer
print(model.predict_from_terms([("C0006826", "C0040329")]))
```

If this outputs something like `[0.9]` (or any other float, if your model hasn't really been trained), then you're good!

3.7 Use Sphinx for Documentation

This guide details some basics on using Sphinx to document Agatha. The goal is to produce a human-readable website on ReadTheDocs.org in the easiest way possible.

3.7.1 Writing Function Descriptions Within Code

I've configured Sphinx to accept Google Docstrings and to parse python3 type-hints. Here's a full example:

```
def parse_predicate_name(predicate_name:str)->Tuple[str, str]:
  """Parses subject and object from predicate name strings.

  Predicate names are formatted strings that follow this convention:
  p:{subj}:{verb}:{obj}. This function extracts the subject and object and
  returns coded-term names in the form: m:{entity}. Will raise an exception if
  the predicate name is improperly formatted.

  Args:
    predicate_name: Predicate name in form p:{subj}:{verb}:{obj}.

  Returns:
    The subject and object formulated as coded-term names.

  """
  typ, sub, vrb, obj = predicate_name.lower().split(":")
```

(continues on next page)

(continued from previous page)

```
    assert typ == PREDICATE_TYPE
    return f"{UMLS_TERM_TYPE}:{sub}", f"{UMLS_TERM_TYPE}:{obj}"
```

Lets break that down. To document a function, first you should write a good function signature. This means that the types for each input and the return value should have associated hints. Here, we have a string input that returns a tuple of two strings. Note, to get type hints for many python standard objects, such as lists, sets, and tuples, you will need to import the `typing` module.

Assuming you've got a good function signature, you can now write a google-formatted docstring. There are certainly more specific formate options than listed here, but at a minimum you should include:

- Single-line summary
- Short description
- Argument descriptions
- Return description

These four options are demonstrated above. Note that this string should occur as a multi-line string (three-quotes) appearing right below the function signature.

Note: at the time of writing, preactically none of the functions follow this guide. If you start modifying the code, try and fill in the backlog of missing docstrings.

3.7.2 Writing Help Pages

Sometimes you will have to write guides that are supplemental to the codebase itself (for instance, this page). To do so, take a look at the `docs` subdirectory from the root of the project. Here, I have setup `docs/help`, and each file within this directory will automatically be included in our online documentation. Furthermore, you can write in either reStructuredText or Markdown. I would recommend Markdown, only because it is simpler. These files must end in either `.rst` or `.md` based on format.

3.7.3 Compiling the Docs

Note that this describes how to build the documentation locally, skip ahead to see how we use ReadTheDocs to automate this process for us.

Assuming the Agatha module has been installed, including the additional modules in `requirements.txt`, you should be good to start compiling. Inside `docs` there is a `Makefile` that is preconfigured to generate the API documentation as well as any extra help files, like this one. Just type `make html` while in `docs` to get that process started.

First, this command will run `sphinx-apidoc` on the `agatha` project in order to extract all functions and docstrings. This process will create a `docs/_api` directory to store all of the intermediate API-generated documentation. Next, it will run `sphinx-build` to compile `html` files from all of the user-supplied and auto-generated `.rst` and `.md` files. The result will be placed in `/docs/build`.

The compilation process may throw a lot of warnings, especially because there are many incorrectly formatted docstrings present in the code that predate our adoption of sphinx and google-docstrings. This is okay as long as the compilation process completes.

3.7.4 Using ReadTheDocs

We host our documentation on ReadTheDocs.org. This service is hooked into our repository and will automatically regenerate our documentation every time we push a commit to master. Behind the scenes this service will build our api documentation read in all of our `.rst` and `.md` files for us. This process will take a while, but the result should appear online after a few minutes.

Updating Dependencies for Read the Docs

The hardest part about ReadTheDocs is getting the remote server to properly install all dependencies needed within the memory and time constraints that come along with using a free 3rd party service. We solve this problem by using a combination of a lightweight conda environment, and heavy use of the mockup function of sphinx autodoc.

Some dependencies, such as protobuf, can only be installed via conda. Additionally, because the conda environment creation process is the first step that ReadTheDocs will perform each build, we also load in our documentation-specific requirements. These modules are specified in `docs/environment.yaml`.

The rest of the dependencies take too long and use too much memory to be installed on ReadTheDocs. At the time of writing we only receive 900 seconds and 500mb of memory in order to build the entire package. Furthermore, many of our dependencies may have version conflicts that can cause unexpected issues that are hard to debug on the remote server. To get around this limitation, we mockup all of our external dependencies when ReadTheDocs builds our project.

When the `READTHEDOCS` environment variable is set to `True`, we make two modifications to our documentation creation process. Firstly, `setup.py` is configured to drop all requirements, meaning that only the Agatha source code itself will be installed. In order to load our source without error, we make the second change in `docs/conf.py`. Here, we set `autodoc_mock_imports` to be a list of all top-level imported modules within Agatha. Unfortunately, some package names are different from their corresponding module names (`pip install faiss_cpu` provides the `faiss` module for instance). *Therefore, the list of imported modules has to be duplicated in `docs/conf.py`.*

Because we are mocking up all of our dependencies, there are some lower-quality documents in places. Specifically, where we use type hints for externally defined classes. Future work could try to selectively enable some modules for better documentation on ReadTheDocs. However, one can always build higher-quality documentation locally by installing the package with all dependencies and running `make html` in `docs/`.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

a

INDEX

A

C

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X