nesta Documentation

nesta

Aug 13, 2021

Contents:

[Archived]

CHAPTER 1

Packages

Nesta's collection of tools for meaty tasks. Any processes that go into production come here first, but there are other good reasons for code to end up here.

1.1 Code and scripts

1.1.1 Meetup

NB: The meetup pipeline will not work until this issue **has been resolved.**

Data collection of Meetup data. The procedure starts with a single country and Meetup category. All of the groups within the country are discovered, from which all members are subsequently retrieved (no personal information!). In order to build a fuller picture, all other groups to which the members belong are retrieved, which may be in other categories or countries. Finally, all group details are retrieved.

The code should be executed in the following order, which reflects the latter procedure:

1) country_groups.py
2) groups_members.py
3) members_groups.py
4) groups_details.py

Each script generates a list of dictionaries which can be ingested by the proceeding script.

Country → Groups

Start with a country (and Meetup category) and end up with Meetup groups.

Groups → Members

Start with Meetup groups and end up with Meetup members.

Members → Groups

Start with Meetup members and end up with Meetup groups.

Groups → Group details

Start with Meetup groups and end up with Meetup group details.

Utils

Common tools between the different data collection points.

1.1.2 Health data

Initially for our project with the Robert Woods Johnson Foundation (RWJF), these procedures outline the data collection of health-specific data.

Collect NIH

Extract all of the NIH World RePORTER data via their static data dump. `N_TABS` outputs are produced in CSV format (concatenated across all years), where `N_TABS` correspondes to the number of tabs in the main table found at:

> https://exporter.nih.gov/ExPORTER_Catalog.aspx

The data is transferred to the Nesta intermediate data bucket.

`get_data_urls`(*tab_index*)
Get all CSV URLs from the `` tab_index`th tab of the main table found at :code:`TOP_URL ``.

> **Parameters** **`tab_index`** (*`int`*) – Tab number (0-indexed) of table to extract CSV URLs from.
>
> **Returns** Title of the tab in the table. hrefs (list): List of URLs pointing to data CSVs.
>
> **Return type** title (str)

`clean_field_name`(*field_name*)
Standardise inconsistently formatted field names, by replacing non-alphanums with single underscores and lowercasing.

`iterrows`(*url*)
Yield rows from the CSV (found at URL `url`) as JSON (well, `dict` objects).

> **Parameters** **`url`** (*`str`*) – The URL at which a zipped-up CSV is found.
>
> **Yields** `dict` object, representing one row of the CSV.

preprocess_nih

Data cleaning / wrangling before ingestion of raw data, specifically:

- Systematically removing generic prefixes using very hard-coded logic.
- Inferring how to correctly deal with mystery question marks, using very hard-coded logic.
- Splitting of strings into arrays as indicated by JSON the ORM,
- CAPS to Camel Case for any string field which isn't VARCHAR(n) < 10
- Dealing consistently with null values
- explicit conversion to datetime of relevant fields

get_json_cols
Return the column names in the ORM which are of JSON type

get_long_text_cols
Return the column names in the ORM which are a text type, (i.e. TEXT or VARCHAR) and if a max length is specified, with max length > 10. The length requirement is because we don't want to preprocess ID-like or code fields (e.g. ISO codes).

get_date_cols
Return the column names in the ORM which are of JSON type

is_nih_null (*value*, *nulls=(''[ ], {}, 'N/A', 'Not Required', 'None')*)
Returns True if the value is listed in the *nulls* argument, or the value is NaN, null or None.

expand_prefix_list
Expand GENERIC_PREFIXES to include integers, and then a large numbers of permutations of additional characters, upper case and title case. From tests, this covers 19 out of 20 generic prefixes from either abstract text or the "PHR" field.

remove_generic_suffixes (*text*)
Iteratively remove any of the generic terms in *expand_prefix_list* from the front of the text, until none remain.

remove_large_spaces (*text*)
Iteratively replace any large spaces or tabs with a single space, until none remain.

replace_question_with_best_guess (*text*)
Somewhere in NiH's backend, they have a unicode processing problem. From inspection, most of the '?' symbols have quite an intuitive origin, and so this function contains the hard-coded logic for inferring what symbol used to be in the place of each '?'.

remove_trailing_exclamation (*text*)
A lot of abstracts end with '!' and then a bunch of spaces.

upper_to_title (*text*, *force_title=False*)
Inconsistently, NiH has fields as all upper case. Convert to titlecase

clean_text (*text*, *suffix_removal_length=100*)
Apply the full text-cleaning procedure.

detect_and_split (*value*)
Split values either by colons or commas. If there are more colons than commas (+1), then colons are used for splitting (this takes into account that NiH name fields are written as 'last_name, first_name; next_last_name, next_first_name'). Otherwise NiH list fields are delimeted by commas.

split_and_clean (*col_value*)
Apply *detect_and_split* and then apply some general cleaning.

`parse_date` (*value*)
Convert to date, unless the value null, or poorly formatted.

`remove_unspecified_unicode` (*value*)
In a very small number of cases, NiH has some pre-processed badly formatted unspecifed unicode characters. The following recipe seems to clean up all of the discovered cases.

`preprocess_row` (*row*, *orm*)
Clean text, split values and standardise nulls, as required.

Parameters

- **`row`** (*`dict`*) – Row of data to clean, that should match the provided ORM.
- **`orm`** (*`SqlAlchemy selectable`*) – ORM from which to infer JSON and text fields.

Process NIH

Data cleaning and processing procedures for the NIH World Reporter data. Specifically, a lat/lon is generated for each city/country; and the formatting of date fields is unified.

1.1.3 NLP Utils

Standard tools for aiding natural language processing.

Preprocess

Tools for preprocessing text.

`tokenize_document` (*text*, *remove_stops=False*, *keep_quasi_numeric=True*)
Preprocess a whole raw document. :param text: Raw string of text. :type text: str :param remove_stops: Flag to remove english stopwords :type remove_stops: bool

Returns List of preprocessed and tokenized documents

`clean_and_tokenize` (*text*, *remove_stops*, *keep_quasi_numeric=False*)
Preprocess a raw string/sentence of text. :param text: Raw string of text. :type text: str :param remove_stops: Flag to remove english stopwords :type remove_stops: bool

Returns Preprocessed tokens.

Return type tokens (list, str)

`filter_by_idf` (*documents*, *lower_idf_limit*, *upper_idf_limit*)
Remove (from documents) terms which are in a range of IDF values.

Parameters

- **`documents`** (*`list`*) – Either a `list` of `str` or a `list` of `list` of `str` to be filtered.
- **`lower_idf_limit`** (*`float`*) – Lower percentile (between 0 and 100) on which to exclude terms by their IDF.
- **`upper_idf_limit`** (*`float`*) – Upper percentile (between 0 and 100) on which to exclude terms by their IDF.

Returns Filtered documents

1.1.4 Geo Utils

Tools for processing of geographical data, such as geocoding.

geocode

Tools for geocoding.

`geocode`

Geocoder using the Open Street Map Nominatim API.

If there are multiple results the first one is returned (they are ranked by importance). The API usage policy allows maximum 1 request per second and no multithreading: https://operations.osmfoundation.org/policies/nominatim/

Parameters **`request_kwargs`** (`dict`) – Parameters for OSM API.

Returns JSON from API response.

`retry_if_not_value_error`(*exception*)

Forces retry to exit if a valueError is returned. Supplied to the 'retry_on_exception' argument in the retry decorator.

Parameters **`exception`** (`Exception`) – the raised exception, to check

Returns False if a ValueError, else True

Return type (bool)

`geocode_dataframe`(*df*)

A wrapper for the geocode function to process a supplied dataframe using the city and country.

Parameters **`df`** (`dataframe`) – a dataframe containing city and country fields.

Returns a dataframe with a 'coordinates' column appended.

`geocode_batch_dataframe`(*df*, *city='city'*, *country='country'*, *latitude='latitude'*, *longitude='longitude'*, *query_method='both'*)

Geocodes a dataframe, first by supplying the city and country to the api, if this fails a second attempt is made supplying the combination using the q= method. The supplied dataframe df is returned with additional columns appended, containing the latitude and longitude as floats.

Parameters

- **`df`** (`pandas.DataFrame`) – input dataframe
- **`city`** (`str`) – name of the input column containing the city
- **`country`** (`str`) – name of the input column containing the country
- **`latitude`** (`str`) – name of the output column containing the latitude
- **`longitude`** (`str`) – name of the output column containing the longitude
- **`query_method`** (`int`) – query methods to attempt: 'city_country_only': city and country only 'query_only': q method only 'both': city, country with fallback to q method

Returns original dataframe with lat and lon appended as floats

Return type (`pandas.DataFrame`)

`generate_composite_key`(*city=None*, *country=None*)

Generates a composite key to use as the primary key for the geographic data.

Parameters

- **city** (*str*) – name of the city
- **country** (*str*) – name of the country

Returns composite key

Return type (str)

1.1.5 Format Utils

Tools for formatting data, such as dates.

datetools

Tools for processing dates in data.

extract_year (*date*)
Use search for 4 digits in a row to identify the year and return as YYYY-01-01.

Parameters **date** (*str*) – The full date string.

Returns integer

extract_date (*date*, *date_format='%Y-%m-%d'*, *return_date_object=False*)
Determine the date format, convert and return in YYYY-MM-DD format.

Parameters **date** (*str*) – the full date string.

Returns Formatted date string.

1.1.6 Decorators

ratelimit

Apply rate limiting at a threshold per second

ratelimit (*max_per_second*)

Parameters **max_per_second** (*float*) – Number of permitted hits per second

schema_transform

Apply a field name transformation to a data output from the wrapped function, such that specified field names are transformed and unspecified fields are dropped. A valid file would be formatted as shown:

{ "tier0_to_tier1":

{ "bad_col": "good_col", "another_bad_col": "another_good_col"

}

}

load_transformer (*filename*)

schema_transform (*filename*)

Parameters **filename** (*str*) – A record-oriented JSON file path mapping field names

Returns Data in the format it was originally passed to the wrapper in, with specified field names transformed and unspecified fields dropped.

`schema_transformer`(*data*, ***, *filename*, *ignore=[]*)

Function version of the schema_transformer wrapper. :param data: the data requiring the schama transformation :type data: dataframe OR list of dicts :param filename: the path to the schema json file :type filename: str :param ignore: optional list of fields, eg ids or keys which shouldn't be dropped :type ignore: list

Returns supplied data with schema applied

1.2 Code auditing

Packages are only accepted if they satisfy our internal auditing procedure:

- **Common sense requirements:**
  - **Either:**
    - The code produces at least one data or model output; **or**
    - The code provides a service which abstracts away significant complexity.
  - There is one unit test for each function or method, which lasts no longer than about 1 minute.
  - Each data or model output is produced from a single function or method, as described in the `__main__` of a specified file.
  - Can the nearest programmer (or equivalent) checkout and execute your tests from scratch?
  - Will the code be used to perform non-project specific tasks?
  - Does the process perform a logical task or fulfil a logical purpose?
- **If the code requires productionising, it satisfies one of the following conditions:**
  a) There is a non-trivial pipeline, which would benefit from formal productionising.
  b) A procedure is foreseen to be reperformed for new contexts with atomic differences in run conditions.
  c) The output is a service which requires a pipeline.
  d) The process is a regular / longitudinal data collection.
- **Basic PEP8 and style requirements:**
  - Docstrings for every exposable class, method and function.
  - Usage in a README.rst or in Docstring at the top of the file.
  - CamelCase class names.
  - Underscore separation of all other variable, function and module names.
  - No glaring programming no-nos.
  - Never use `print`: opt for `logging` instead.
- **Bureaucratic requirements:**
  - A requirements file*.
  - The README file specifies the operating system and python version.

CHAPTER 2

Production

Nesta's production system is based on Luigi pipelines, and are designed to be entirely run on AWS via the batch service. The main Luigi server runs on a persistent EC2 instance. Beyond the well documented Luigi code, the main features of the nesta production system are:

- `luigihacks.autobatch`, which facilates a managed `Luigi.Task` which is split, batched and combined in a single step. Currently only synchronous jobs are accepted. Asynchonous jobs (where downstream `Luigi.Task` jobs can be triggered) are a part of a longer term plan.
- `scripts.nesta_prepare_batch` which zips up the batchable with the specified environmental files and ships it to AWS S3.
- `scripts.nesta_docker_build` which builds a specified docker environment and ships it to AWS ECS.

2.1 How to put code into production at nesta

If you're completely new, check out our training slides. In short, the steps you should go through when building production code are to:

1. Audit the package code, required to pass all auditing tests
2. Understand what environment is required
3. Write a Dockerfile and docker launch script for this under scripts/docker_recipes
4. Build the Docker environment (run: docker_build <recipe_name> from any directory)
5. Build and test the batchable(s)
6. Build and test a Luigi pipeline
7. [. . .] Need to have steps here which estimate run time cost parameters. Could use tests.py to estimate this. [. . .]
8. Run the full chain

2.2 Code and scripts

2.2.1 Routines

All of our pipelines, implemented as Luigi routines. Some of these pipelines (at least partly) rely on batch computing (via AWS Batch), where the 'batched' scripts (*run.py* modules) are described in `core.batchables`. Other than `luigihacks.autobatch`, which is respectively documented, the routine procedure follows the core Luigi documentation.

Examples

Examples of Luigi routines, from which all other nesta production routines can be built. Currently we have examples of routines with S3 and database (MySQL) IO, and routines which are entirely batched.

We'd recommend reading Spotify's Luigi documentation, and also checking the *Luigi Hacks* documentation which contains modified Luigi modules which (who knows) one day we will suggest as pull requests.

S3 Example

An example of building a pipeline with S3 Targets

class InputData(**args*, ***kwargs*)
Bases: `luigi.task.ExternalTask`

Dummy task acting as the single input data source

output()
Points to the S3 Target

class SomeTask(**args*, ***kwargs*)
Bases: `luigi.task.Task`

An intermediate task which increments the age of the muppets by 1 year.

Parameters **date** (*datetime*) – Date used to label the outputs

date = <luigi.parameter.DateParameter object>

requires()
Gets the input data (json containing muppet name and age)

output()
Points to the S3 Target

run()
Increments the muppets' ages by 1

class FinalTask(**args*, ***kwargs*)
Bases: `luigi.task.Task`

The root task, which adds the surname 'Muppet' to the names of the muppets.

Parameters **date** (*datetime*) – Date used to label the outputs

date = <luigi.parameter.DateParameter object>

requires()
Get data from the intermediate task

output ()
Points to the S3 Target

run ()
Appends 'Muppet' the muppets' names

Database example

An example of building a pipeline with database Targets

class InputData (**args*, ***kwargs*)
Bases: `luigi.task.Task`

Dummy task acting as the single input data source.

Parameters

- **date** (*datetime*) – Date used to label the outputs
- **db_config** – (dict) The input database configuration

date = <luigi.parameter.DateParameter object>

db_config = <luigi.parameter.DictParameter object>

output ()
Points to the input database target

run ()
Example of marking the update table

class SomeTask (**args*, ***kwargs*)
Bases: `luigi.task.Task`

Task which increments the age of the muppets, by first selecting muppets with an age less than *max_age*.

Parameters

- **date** (*datetime*) – Date used to label the outputs
- **max_age** (*int*) – Maximum age of muppets to select from the database
- **in_db_config** – (dict) The input database configuration
- **out_db_config** – (dict) The output database configuration

date = <luigi.parameter.DateParameter object>

max_age = <luigi.parameter.IntParameter object>

in_db_config = <luigi.parameter.DictParameter object>

out_db_config = <luigi.parameter.DictParameter object>

requires ()
Gets the input database engine

output ()
Points to the output database engine

run ()
Increments the muppets' ages by 1

class RootTask(**args*, ***kwargs*)

Bases: `luigi.task.WrapperTask`

A dummy root task, which collects the database configurations and executes the central task.

Parameters **date** (*datetime*) – Date used to label the outputs

date = <luigi.parameter.DateParameter object>

requires()

Collects the database configurations and executes the central task.

arXiv data (technical research)

Data collection and processing pipeline for arXiv data, principally for the arXlive platform. This pipeline orchestrates the collection of arXiv data, enrichment (via MAG and GRID), topic modelling, and novelty (lolvelty) measurement.

CORDIS (EU funded research)

Generic pipeline (i.e. not project specific) to collect all CORDIS data, discovering all entities by crawling an unofficial API.

Crunchbase (private sector companies)

NB: The Crunchbase pipeline may not work until this issue **has been resolved.**

Data collection and processing pipeline of Crunchbase data, principally for the healthMosaic platform.

EURITO (piping data to Elasticsearch)

Pipeline specific to EURITO for piping existing data to Elasticsearch. A recent "EU" cut of patstat data is transferred from the "main" patstat database, to Nesta's central database.

Gateway to Research (UK publicly funded research)

Generic pipeline (i.e. not project specific) to collect all GtR data, discovering all entities by crawling the official API. The routine then geocodes and loads data to MYSQL.

NiH (health research)

Data collection and processing pipeline of NiH data, principally for the healthMosaic platform.

Meetup (social networking data)

NB: The Meetup pipeline will not work until this issue **has been resolved.**

Data collection and processing pipeline of Meetup data, principally for the healthMosaic platform.

2.2.2 Batchables

Packets of code to be batched by `core.routines` routines. Each packet should sit in it's own directory, with a file called `run.py`, containing a 'main' function called `run()` which will be executed on the AWS batch system.

Each *run.py* should expect an environment parameter called `BATCHPAR_outfile` which should provide information on the output location. Other input parameters should be prefixed with `BATCHPAR_`, as set in `core.routines` routine.

Data / project specific batchables

Example

There are two batchable examples listed here. The first is a module which will be run if you try executing the *batch_example* luigi routine. The second is purely meant as a template, if you are learning the design pattern for nesta's luigi batchables.

run.py (batch_example)

The batchable for the `routines.examples.batch_example`, which simply increments a muppet's age by one unit.

run()
    Gets the name and age of the muppet, and increments the age. The result is transferred to S3.

arXiv data (technical research)

CORDIS (EU-funded research)

Crunchbase data (private companies)

NB: The Crunchbase pipeline may not work until this issue **has been resolved.**

Batchables for the collection and processing of Crunchbase data. As documented under *packages* and *routines*, the pipeline is executed in the following order (documentation for the *run.py* files is given below, which isn't super-informative. You're better off looking under packages and routines).

The data is collected from proprietary data dumps, parsed into MySQL (tier 0) and then piped into Elasticsearch (tier 1), post-processing.

EURITO

Batchables for processing data (which has already been collected elsewhere within this codebase) for the EURITO project. All of these batchables pipe the data into an Elasticsearch database, which is then cloned by EURITO.

GtR (UK publicly funded research)

Batchable tools for collecting and processing GtR data. As documented under packages and routines, the pipeline is executed in the following order (documentation for the run.py files is given below, which isn't super-informative. You're better off looking under packages and routines).

The data is collected by traversing the graph exposed by the GtR API, and is parsed into MySQL (tier 0). There is a further module for directly generating document embeddings of GtR project descriptions, which can be used for finding topics.

NiH data (health research)

Batchables for the collection and processing of NiH data. As documented under *packages* and *routines*, the pipeline is executed in the following order (documentation for the *run.py* files is given below, which isn't super-informative. You're better off looking under packages and routines).

The data is collected from official data dumps, parsed into MySQL (tier 0) and then piped into Elasticsearch (tier 1), post-processing.

Meetup (social networking / knowledge exchange)

NB: The meetup pipeline will not work until this issue **has been resolved.**

Batchables for the Meetup data collection pipeline. As documented under *packages* and *routines*, the pipeline is executed in the following order (documentation for the *run.py* files is given below, which isn't super-informative. You're better off looking under packages and routines).

The `topic_tag_elasticsearch` module is responsible for piping data to elasticsearch, as well as apply topic tags and filtering small groups out of the data.

General-purpose batchables

Bulk geocoding

Natural Language Processing

Batchable utilities for NLP. Note that modules prefixed with `[AUTOML]` are designed to be launched by `AutoMLTask`, and those with the addition * (i.e. `[AUTOML*]`) are the designed to be the final task in an `AutoMLTask` chain (i.e. they provide a 'loss').

[AutoML*] run.py (corex_topic_model)

Generate topics based on the CorEx algorithm. Loss is calculated from the total correlation explained.

`run``()`

[AutoML] run.py (tfidf)

Applies TFIDF cuts to a dataset via environmental variables lower_tfidf_percentile and upper_tfidf_percentile.

`chunker``(`*_transformed*, *n_chunks*`)`

Yield chunks from a numpy array.

Parameters

- **`_transformed`** (*`np.array`*) – Array to split into chunks.
- **`n_chunks`** (*`int`*) – Number of chunks to split the array into.

Yields chunk (np.array)

run()

[AutoML] vectorizer (run.py)

Vectorizes (counts or binary) text data, and applies basic filtering of extreme term/document frequencies.

term_counts(*dct*, *row*, *binary=False*)
Convert a single single document to term counts via a gensim dictionary.

Parameters

- **dct** (*Dictionary*) – Gensim dictionary.
- **row** (*str*) – A document.
- **binary** (*bool*) – Binary rather than total count?

Returns dict of term id (from the Dictionary) to term count.

optional(*name*, *default*)
Defines optional env fields with default values

merge_lists(*list_of_lists*)
Join a lists of lists into a single list. Returns an empty list if the input is not a list, which is expected to happen (from the ngrammer) if no long text was found

run()

Novelty

Batchables for calculating measures of "novelty".

2.2.3 ORMs

`SQLAlchemy` ORMs for the routines, which allows easy integration of testing (including automatic setup of test databases and tables).

Meetup

class Group(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

id

urlname

category_name

category_shortname

city

country

created

description

lat

lon

members

name

topics

category_id

country_name

timestamp

class GroupMember(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Note: no foreign key constraint, since unknown groups will be found in the member expansion phase

group_id

group_urlname

member_id

NIH schema

The schema for the World RePORTER data.

getattr_(*entity*, *attribute*)

Either unpack the attribute from every item in the entity if the entity is a list, otherwise just return the attribute from the entity. Returns None if the entity is either None or empty.

class Projects(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

application_id

activity

administering_ic

application_type

arra_funded

award_notice_date

base_core_project_num

budget_start

budget_end

cfda_code

core_project_num

ed_inst_type

foa_number

full_project_num

funding_ics

`funding_mechanism`

`fy`

`ic_name`

`org_city`

`org_country`

`org_dept`

`org_district`

`org_duns`

`org_fips`

`org_ipf_code`

`org_name`

`org_state`

`org_zipcode`

`phr`

`pi_ids`

`pi_names`

`program_officer_name`

`project_start`

`project_end`

`project_terms`

`project_title`

`serial_number`

`study_section`

`study_section_name`

`suffix`

`support_year`

`direct_cost_amt`

`indirect_cost_amt`

`total_cost`

`subproject_id`

`total_cost_sub_project`

`nih_spending_cats`

`abstract`

`publications`

`patents`

`clinicalstudies`

abstract_text

patent_ids

patent_titles

pmids

clinicaltrial_ids

clinicaltrial_titles

class Abstracts(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

application_id

abstract_text

class Publications(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

pmid

author_name

affiliation

author_list

country

issn

journal_issue

journal_title

journal_title_abbr

journal_volume

lang

page_number

pub_date

pub_title

pub_year

pmc_id

class Patents(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

patent_id

patent_title

project_id

patent_org_name

class LinkTables(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

pmid

project_number

class ClinicalStudies(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

clinicaltrials_gov_id

core_project_number

study

study_status

class PhrVector(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Document vectors for NiH Public Health Relevance (PHR) statements.

application_id

vector

class AbstractVector(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Document vectors for NiH abstracts.

application_id

vector

class TextDuplicate(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Link table to describe for NiH text-field duplicates, which probably imply that projects are related, either formally (if weight > 0.8 they are normally almost exact duplicates of each other) or contextually (if weight > 0.5 it is normally in the same general subject area).

The cut-off for inclusion in this table is a weight of 0.5, because the core interest for using this method is to identify texts which are near duplicates, since texts which are contextually similar can also be found by other metrics (topic modelling, etc) and there can be some weird side-effects of using BERT for this; e.g. finding texts with a similar writing style rather than topic.

application_id_1

application_id_2

text_field

weight

2.2.4 Ontologies and schemas

Tier 0

Raw data collections ("tier 0") in the production system do not adhere to a fixed schema or ontology, but instead have a schema which is very close to the raw data. Modifications to field names tend to be quite basic, such as lowercase and removal of whitespace in favour of a single underscore.

Tier 1

Processed data ("tier 1") is intended for public consumption, using a common ontology. The convention we use is as follows:

- Field names are composed of up to three terms: a `firstName`, `middleName` and `lastName`
- Each term (e.g. `firstName`) is written in lowerCamelCase.
- `firstName` terms correspond to a restricted set of basic quantities.
- `middleName` terms correspond to a restricted set of modifiers (e.g. adjectives) which add nuance to the `firstName` term. Note, the special `middleName` term `of` is reserved as the default value in case no `middleName` is specified.
- `lastName` terms correspond to a restricted set of entity types.

Valid examples are `date_start_project` and `title_of_project`.

Tier 0 fields are implictly excluded from tier 1 if they are missing from the `schema_transformation` file. Tier 1 schema field names are applied via *nesta.packages.decorator.schema_transform*

Tier 2

Although not-yet-implemented, the tier 2 schema is reserved for future graph ontologies. Don't expect any changes any time soon!

2.2.5 Elasticsearch mappings

Our methodology for constructing Elasticsearch mappings is described here. It is intended to minimise duplication of efforts and enforce standardisation when referring to a common dataset whilst being flexible to individual project needs. It is implied in our framework that a single `dataset` can be used across many projects, and each project is mapped to a single `endpoint`. It is useful to start by looking at the structure of the `nesta/core/schemas/tier_1/mappings/` directory:

```
.
├── datasets
│   ├── arxiv_mapping.json
│   ├── companies_mapping.json
│   ├── cordis_mapping.json
│   ├── gtr_mapping.json
│   ├── meetup_mapping.json
│   ├── nih_mapping.json
│   └── patstat_mapping.json
├── defaults
│   └── defaults.json
└── endpoints
    ├── arxlive
    │   └── arxiv_mapping.json
    ├── eurito
    │   ├── arxiv_mapping.json
    │   ├── companies_mapping.json
    │   └── patstat_mapping.json
    └── health-scanner
        ├── aliases.json
        ├── config.yaml
        ├── nih_mapping.json
        └── nulls.json
```

Firstly we consider `defaults/defaults.json` which should contain all default fields for all mappings - for example standard analyzers and dynamic strictness. We might also consider putting global fields there.

Next consider the `datasets` subdirectory. Each mapping file in here should contain the complete `mappings` field for the respective dataset. The naming convention `<dataset>_mapping.json` is a hard requirement, as `<dataset>` will map to the index for this `dataset` at any given `endpoint`.

Finally consider the `endpoints` subdirectory. Each sub-subdirectory here should map to any `endpoint` which requires changes beyond the `defaults` and `datasets` mappings. Each mapping file within each `endpoint` sub-subdirectory (e.g. `arxlive` or `health-scanner`) should satisfy the same naming convention (`<dataset>_mapping.json`). All conventions here are also consistent with the `elasticsearch.yaml` configuration file (to see this configuration, you will need to clone the repo and follow these steps to unencrypt the config), which looks a little like this:

```
## The following assumes the AWS host endpoing naming convention:
## {scheme}://search-{endpoint}-{id}.{region}.es.amazonaws.com
defaults:
  scheme: https
  port: 443
  region: eu-west-2
  type: _doc
endpoints:
  # -------------------------------
  # <AWS endpoint domain name>:
  #   id: <AWS endpoint UUID>
  #   <default override key>: <default override value>  ## e.g.: scheme, port, region,
↪ _type
  #   indexes:
  #     <index name>: <incremental version number>  ## Note: defaults to <index name>_
↪dev in testing mode
  # -------------------------------
  arxlive:
    id: <this is a secret>
    indexes:
      arxiv: 4
  # -------------------------------
  health-scanner:
    id: <this is a secret>
    indexes:
      nih: 6
      companies: 5
      meetup: 4
... etc ...
```

Note that for the `health-scanner` endpoint, `companies` and `meetup` will be generated from the `datasets` mappings, as they are not specified under the `endpoints/health-scanner` subdirectory. Also note that `endpoints` sub-directories do not need to exist for each `endpoint` to be generated: the mappings will simply be generated from the dataset defaults. For example, a new endpoint `general` can be generated from the DAPS codebase using the above, even though there is no `endpoints/general` sub-subdirectory.

Individual `endpoints` can also specify `aliases.json` which harmonises field names across datasets for specific endpoints. This uses a convention as follows:

```
{
    #...the convention is...
    "<new field name>": {
        "<dataset 1>": "<old field name 1>",
        "<dataset 2>": "<old field name 2>",
```

(continues on next page)

(continued from previous page)

```
        "<dataset 3>": "<old field name 3>"
    },
    #...an example is...
    "city": {
        "companies": "placeName_city_organisation",
        "meetup": "placeName_city_group",
        "nih": "placeName_city_organisation"
    },
    #...etc...#
}
```

By default, this applies (what Joel calls) a "soft" alias, which is an Elasticsearch alias, however by specifying `hard-alias=true` in `config.yaml` (see `health-scanner` above), the alias is instead applied directly (i.e. field names are physically replaced, not aliased).

You will also notice the `nulls.json` file in the `health-scanner` endpoint. This is a relatively experimental feature for automatically nullifying values on ingestion through ElasticsearchPlus, in lieu of proper exploratory data analysis. The logic and format for this is documented here.

Mapping construction hierarchy

Each mapping is constructed by overriding nested fields using the `defaults` `datasets` and `endpoints`, in that order (i.e. `endpoints` override nested fields in `datasets`, and `datasets` override those in `defaults`). If you would like to "switch off" a field from the `defaults` or `datasets` mappings, you should set the value of the nested field to `null`. For example:

```
{
    "mappings": {
        "_doc": {
            "dynamic": "strict",
            "properties": {
                "placeName_zipcode_organisation": null
            }
        }
    }
}
```

will simply "switch off" the field `placeName_zipcode_organisation`, which was specified in `datasets`.

The logic for the mapping construction hierarchy is demonstrated in the respective `orms.orm_utils.get_es_mapping` function:

```
def get_es_mapping(dataset, endpoint):
    '''Load the ES mapping for this dataset and endpoint,
    including aliases.

    Args:
        dataset (str): Name of the dataset for the ES mapping.
        endpoint (str): Name of the AWS ES endpoint.
    Returns:
        :obj:`dict`
    '''
    mapping = _get_es_mapping(dataset, endpoint)
    _apply_alias(mapping, dataset, endpoint)
    _prune_nested(mapping)  # prunes any nested keys with null values
    return mapping
```

Integrated tests

The following `pytest` tests are made (and triggered on PR via travis):

- `aliases.json` files are checked for consistency with available `datasets`.
- All mappings for each in `datasets` and `endpoints` are fully generated, and tested for compatibility with the schema transformations (which are, in turn, checked against the valid ontology in `ontology.json`).

Features in DAPS2

- The index version (e.g. `'arxiv': 4` in `elasticsearch.yaml`) will be automatically generated from semantic versioning and the git hash in DAPS2, therefore the `indexes` field will consolidate to an itemised list of indexes.
- The mappings under `datasets` will be automatically generated from the open ontology which will be baked into the tier-0 schemas. This will render `schema_transformations` redundant.
- Elasticsearch components will be factored out of `orm_utils`.

2.2.6 Luigi Hacks

Modifications and possible future contributions to the Luigi module.

autobatch

batchclient

NOTE: overwhelmingly based on this, where the following documentation has been directly lifted. The main difference to the latter, is that AWS jobs are submitted via `**kwargs` in order to allow more flexibility (and probably more future-proofing if new parameters are added to boto3).

AWS Batch wrapper for Luigi

From the AWS website:

> AWS Batch enables you to run batch computing workloads on the AWS Cloud.
>
> Batch computing is a common way for developers, scientists, and engineers to access large amounts of compute resources, and AWS Batch removes the undifferentiated heavy lifting of configuring and managing the required infrastructure. AWS Batch is similar to traditional batch computing software. This service can efficiently provision resources in response to jobs submitted in order to eliminate capacity constraints, reduce compute costs, and deliver results quickly.

See AWS Batch User Guide for more details.

To use AWS Batch, you create a jobDefinition JSON that defines a docker run command, and then submit this JSON to the API to queue up the task. Behind the scenes, AWS Batch auto-scales a fleet of EC2 Container Service instances, monitors the load on these instances, and schedules the jobs.

This boto3-powered wrapper allows you to create Luigi Tasks to submit Batch `jobDefinition``s. You can either pass a dict (mapping directly to the ``jobDefinition` JSON) OR an Amazon Resource Name (arn) for a previously registered `jobDefinition`.

Requires:

- boto3 package
- Amazon AWS credentials discoverable by boto3 (e.g., by using `aws configure` from awscli)

- An enabled AWS Batch job queue configured to run on a compute environment.

Written and maintained by Jake Feala (@jfeala) for Outlier Bio (@outlierbio)

`exception BatchJobException`
Bases: `Exception`

`class BatchClient`(*poll_time=10*, ***kwargs*)
Bases: `object`

`get_active_queue`()
Get name of first active job queue

`get_job_id_from_name`(*job_name*)
Retrieve the first job ID matching the given name

`get_job_status`(*job_id*)
Retrieve task statuses from ECS API

Parameters **(str)** (*job_id*) – AWS Batch job uuid

Returns one of {SUBMITTED|PENDING|RUNNABLE|STARTING|RUNNING|SUCCEEDED|FAILED}

`get_logs`(*log_stream_name*, *get_last=50*)
Retrieve log stream from CloudWatch

`submit_job`(***kwargs*)
Wrap submit_job with useful defaults

`terminate_job`(***kwargs*)
Wrap terminate_job

`hard_terminate`(*job_ids*, *reason*, *iattempt=0*, ***kwargs*)
Terminate all jobs with a hard(ish) exit via an Exception. The function will also wait for jobs to be explicitly terminated

`wait_on_job`(*job_id*)
Poll task status until STOPPED

`register_job_definition`(*json_fpath*)
Register a job definition with AWS Batch, using a JSON

misctools

A collection of miscellaneous tools.

`get_config`(*file_name*, *header*, *path='core/config/'*)
Get the configuration from a file in the luigi config path directory, and convert the key-value pairs under the config `header` into a *dict*.

Parameters

- **file_name** (*str*) – The configuation file name.
- **header** (*str*) – The header key in the config file.

Returns `dict`

`get_paths_from_relative`(*relative=1*)
A helper method for within *find_filepath_from_pathstub*. Prints all file and directory paths from a relative number of 'backward steps' from the current working directory.

find_filepath_from_pathstub(*path_stub*)
Find the full path of the 'closest' file (or directory) to the current working directory ending with `path_stub`. The *closest* file is determined by starting forwards of the current working directory. The algorithm is then repeated by moving the current working directory backwards, one step at a time until the file (or directory) is found. If the HOME directory is reached, the algorithm raises `FileNotFoundError`.

Parameters **path_stub** (*str*) – The partial file (or directory) path stub to find.

Returns The full path to the partial file (or directory) path stub.

f3p(*path_stub*)
Shortened name for coding convenience

load_yaml_from_pathstub(*pathstub*, *filename*)
Basic wrapper around *find_filepath_from_pathstub* which also opens the file (assumed to be yaml).

Parameters

- **pathstub** (*str*) – Stub of filepath where the file should be found.
- **filename** (*str*) – The filename.

Returns The file contents as a json-like object.

load_batch_config(*luigi_task*, *additional_env_files=[]*, ***overrides*)
Load default luigi batch parametes, and apply any overrides if required. Note that the usage pattern for this is normally `load_batch_config(self, additional_env_files, **overrides)` from within a luigi Task, where `self` is the luigi Task.

Parameters

- **luigi_task** (*luigi.Task*) – Task to extract test and date parameters from.
- **additional_env_files** (*list*) – List of files to pass directly to the batch local environment.
- **overrides** (**kwargs) – Any overrides or additional parameters to pass to the batch task as parameters.

Returns Batch configuration paramaters, which can be expanded as **kwargs in BatchTask.

Return type config (dict)

extract_task_info
Extract task name and generate a routine id from a luigi task, from the date and test fields.

Parameters **luigi_task** (*luigi.Task*) – Task to extract test and date parameters from.

Returns Test flag, and routine ID for this task.

Return type {test, routine_id} (tuple)

bucket_keys
Get all keys in an S3 bucket.

Parameters **bucket_name** (*str*) – Name of a bucket to query.

Returns Set of keys

Return type keys (set)

mysqldb

NOTE: overwhelmingly based on this2, where the following documentation has been directly lifted. The main difference to the latter, is that `**cnx_kwargs` in the constructor can accept *port* as a key.

make_mysql_target (*luigi_task*, *mysqldb_env='MYSQLDB'*)
Generate a MySQL target for a luigi Task, based on the Task's `date` and `test` parameters, and indicated configuration file.

Parameters

- **luigi_task** (*luigi.Task*) – Task to extract test and date parameters from.
- **mysqldb_env** (*str*) – Environmental variable storing the path to MySQL config.

Returns target (MySqlTarget)

class MySqlTarget (*host*, *database*, *user*, *password*, *table*, *update_id*, ***cnx_kwargs*)
Bases: `luigi.target.Target`

Target for a resource in MySql.

marker_table = 'table_updates'

touch (*connection=None*)
Mark this update as complete.

IMPORTANT, If the marker table doesn't exist, the connection transaction will be aborted and the connection reset. Then the marker table will be created.

exists (*connection=None*)
Returns `True` if the `Target` exists and `False` otherwise.

connect (*autocommit=False*)

create_marker_table ()
Create marker table if it doesn't exist.

Using a separate connection since the transaction might have to be reset.

s3

A more recent implementation of AWS S3 support, stolen from: https://gitlab.com/ced/s3_helpers/blob/master/luigi_s3_target.py, but instead using modern boto3 commands.

merge_dicts (**dicts*)
Merge dicts together, with later entries overriding earlier ones.

parse_s3_path (*path*)
For a given S3 path, return the bucket and key values

class S3FS (***kwargs*)
Bases: `luigi.target.FileSystem`

exists (*path*)
Return true if S3 key exists

remove (*path*, *recursive=True*)
Remove a file or directory from S3

mkdir (*path*, *parents=True*, *raise_if_exists=False*)
Create directory at location `path`

Creates the directory at `path` and implicitly create parent directories if they do not already exist.

Parameters

- **path** (*str*) – a path within the FileSystem to create as a directory.

- **parents** (*bool*) – Create parent directories when necessary. When parents=False and the parent directory doesn't exist, raise luigi.target.MissingParentDirectory
- **raise_if_exists** (*bool*) – raise luigi.target.FileAlreadyExists if the folder already exists.

isdir (*path*)

Return `True` if the location at `path` is a directory. If not, return `False`.

Parameters **path** (*str*) – a path within the FileSystem to check as a directory.

Note: This method is optional, not all FileSystem subclasses implements it.

listdir (*path*)

Return a list of files rooted in path.

This returns an iterable of the files rooted at `path`. This is intended to be a recursive listing.

Parameters **path** (*str*) – a path within the FileSystem to list.

Note: This method is optional, not all FileSystem subclasses implements it.

copy (*path*, *dest*)

Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only single file copying but S3Client copies either a file or a directory as required.

move (*path*, *dest*)

Move a file, as one would expect.

du (*path*)

class S3Target (*path*, *s3_args={}*, ***kwargs*)

Bases: `luigi.target.FileSystemTarget`

fs = None

open (*mode='rb'*)

Open the FileSystem target.

This method returns a file-like object which can either be read from or written to depending on the specified mode.

Parameters **mode** (*str*) – the mode *r* opens the FileSystemTarget in read-only mode, whereas *w* will open the FileSystemTarget in write mode. Subclasses can implement additional options.

class AtomicS3File (*path*, *s3_obj*, ***kwargs*)

Bases: `luigi.target.AtomicLocalFile`

move_to_final_destination ()

2.2.7 Scripts

A set of helper scripts for the batching system.

Note that this directory is required to sit in *$PATH*. By convention, all executables in this directory start with *nesta_* so that our developers know where to find them.

nesta_prepare_batch

Collect a batchable `run.py` file, including dependencies and an automaticlly generated requirements file; which is all zipped up and sent to AWS S3 for batching. This script is executed automatically in `luigihacks.autobatch.AutoBatchTask.run`.

Parameters:

- **BATCHABLE_DIRECTORY**: The path to the directory containing the batchable `run.py` file.
- **ARGS**: Space-separated-list of files or directories to include in the zip file, for example imports.

nesta_docker_build

Build a docker environment and register it with the AWS ECS container repository.

Parameters:

- **DOCKER_RECIPE**: A docker recipe. See `docker_recipes/` for a good idea of how to build a new environment.

2.2.8 Elasticsearch

The following steps will take you through setting up elasticsearch on an EC2 instance.

Launch the EC2 instance and ssh in so the following can be installed:

docker

```
sudo yum install docker -y
```

docker-compose

```
curl -L https://github.com/docker/compose/releases/download/1.22.0/
docker-compose-\`uname -s\` - \`uname -m\` -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose              sudo ln -s /usr/local/bin/
docker-compose /usr/bin/docker-compose
```

more info: https://github.com/docker/compose/releases

docker permissions

```
sudo usermod -a -G docker $USER
```

more info: https://techoverflow.net/2017/03/01/solving-docker-permission-denied-while-trying-to-connect-to-the-docker-daemon-sock

vm.max_map_count

set permanantly in */etc/sysctl.conf* by adding the following line: `vm.max_map_count=262144`

more info: https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html

python 3.6

`sudo yum install python36 -y`

The machine now needs to be rebooted `sudo reboot now`

Docker

the `docker-compose.yml` needs to include ulimits settings::

ulimits:

memlock: soft: -1 hard: -1

nofile: soft: 65536 hard: 65536

Recipes for http or https clusters can be found in: `nesta/core/scripts/elasticsearch`

There is also an EC2 AMI for a http node stored in the London region: `elasticsearch node London vX`

Reindexing data from a remote cluster

- reindex permissions need to be set in the new cluster's *elasticsearch.yml*
- if the existing cluster is AWS hosted ES the ip address needs to be added to the security settings
- follow this guide: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-reindex.html#reindex-from-remote
- *index* and *query* do not need to be supplied
- if reindexing from AWS ES the port should be 443 for https. This is mandatory in the json sent to the reindexing api

2.2.9 Containerised Luigi

Requirements

To containerise a pipeline a few steps are required:

- All imports must be absolute, ie `from nesta.` packages, core etc
- Once testing is complete the code should be committed and pushed to github, as this prevents the need to use local build options
- If building and running locally, Docker must be installed on the local machine and given enough RAM in the settings to run the pipeline
- Any required configuration files must be in `nesta.core.config` ie luigi and mysql config files, any API keys. **This directory is ignored but check before committing**

Build

The build uses a multi-stage Dockerfile to reduce the size of the final image:

1. Code is git cloned and requirements are pip installed into a virtual environment
2. The environment is copied into the second image

From the root of the repository: `docker build -f docker/Dockerfile -t name:tag .`

Where `name` is the name of the created image and `tag` is the chosen tag. Eg `arxlive:latest`. This just makes the run step easier rather than using the generated id

The two stage build will normally just rebuild the second stage pulling in new code only. If a full rebuild is required, eg after requirements.txt has changed then include: `--no-cache`

Python version defaults to 3.6 but can be set during build by including the flag: `--build-arg python-version=3.7`

Tag defaults to *dev* but this can be overridden by including the flag: `--build-arg GIT_TAG=0.3` a branch name also works `--build-arg GIT_TAG=my-branch`

Work from a branch or locally while testing. Override the target branch from Github using the method above, or use the commented out code in the Dockerfile to switch to build from local files. **Don't commit this change!**

Run

As only one pipeline runs in the container the `luigid` scheduler is not used.

There is a `docker-compose` file for arXlive which mounts your local ~.aws folder for AWS credentials as this outside docker's context:

```
docker-compose -f docker/docker-compose.yml run luigi --module module_path
params
```

Where:

- `docker-compose.yml` is the docker-compose file containing the image: `image_name:tag` from the build
- `module_path` is the full python path to the module
- `params` are any other params to supply as per normal, ie `--date` `--production` etc

Eg: `docker-compose -f docker/docker-compose-arxlive-dev.yml run luigi --module nesta.core.routines.arxiv.arxiv_iterative_root_task RootTask --date 2019-04-16`

This could be adapted for each pipeline, or alternatively run with the volume specified with `-v`

```
docker run -v ~/.aws:/root/.aws:ro name:tag --module ...
```

Where `name` is the name of the created image and `tag` is the chosen tag. Eg `arxlive:latest` `--module ...` onwards contains the arguments you would pass to Luigi.

Scheduling

1. Create an executable shell script in `nesta.core.scripts` to launch docker-compose with all the necessary parameters eg: production
2. Add a cron job to the shell script (there are some good online cron syntax checking tools, if needed)
3. Set the cron job to run every few minutes while testing and check the logs with `docker logs mycontainterhash --tail 50`. Obtain the hash using `docker ps`
4. It will run logged in as the user who set it up but there still may still be some permissions issues to deal with

Currently scheduled

arXlive:

- A shell script to launch docker-compose for arXlive is set up to run in a cron job on user `russellwinch`
- This is scheduled for Sunday-Thursday at 0300 GMT. arXiv is updated on these days at 0200 GMT
- Logs are just stored in the container, use `docker logs container_name` to view

Important points

- Keep any built images secure, they contain credentials
- You need to rebuild if code has changed
- As there is no central scheduler there is nothing stopping you from running the task more than once at the same time, by launching the container multiple times
- The graphical interface is not enabled without the scheduler

Debugging

If necessary, it's possible to debug inside the container, but the `endpoint` needs to be overridden with `bash`:

```
docker run --entrypoint /bin/bash -itv ~/.aws:/root/.aws:ro image_name:tag
```

Where `image_name:tag` is the image from the build step This includes the mounting of the .aws folder

Almost nothing is installed (not even vi!!) other than Python so `apt-get update` and then `apt-get install` whatever you need

Todo

A few things are sub-optimal:

- The container runs on the prod box rather than in the cloud in ECS
- Credentials are held in the container and local AWS config is required, this is the cause of the above point
- Due to the Nesta monorepo everything is pip installed, making a large container size with many unrequired packages. Pipeline specific requirements should be considered
- As logs are stored in the old containers they are kept until the next run where they are pruned and the logs are lost. Add a method of getting the logs to the host logger and record centrally
- In the arXlive pipeline there are at least 500 calls to the MAG API each run as the process tries to pick up new title matches on existing articles. As the API key only allows 10,000 calls per month this is currently OK with the schedule as it is but could possibly go over at some point

CHAPTER 3

FAQ

3.1 Where is the data?

As a general rule-of-thumb, our data is always stored in the London region (`eu-west-2`), in either RDS (tier-0, MySQL) or Elasticsearch (tier-1). For the EURITO project we also use Neo4j (tier-1), and in the distant future we will use Neo4j for tier-2 (i.e. a knowledge graph).

3.2 Why don't you use Aurora rather than MySQL?

Aurora is definitely cheaper for stable production and business processes but not for research and development. You are charged for every byte of data you have *ever* consumed. This quickly spirals out-of-control for big-data development. Maybe one day we'll consider migrating back, once the situation stabilises.

3.3 Where are the production machines?

Production machines (EC2) run in Ohio (us-east-2).

3.4 Where is the latest config?

We use `git-crypt` to encrypt our configuration files whilst allowing them to be versioned in git (meaning that we can also rollback configuration). To unlock the configuration encryption, you should install `git-crypt`, then run `bash install.sh` from the project root, and finally unlock the configuration using the key found here.

3.5 Where do I start with Elasticsearch?

All Elasticsearch indexes (aka "databases" to the rest of the world), mappings (aka "schemas") and whitelisting can be found here.

I'd recommend using PostMan for spinning up and knocking down indexes. Practice this on a new cluster (which you can spin up from the above link), and then practice `PUT`, `POST` and `DELETE` requests to `PUT` an index (remember: "database") with a mapping ("schema"), inserting a "row" with `POST` and then deleting the index with `DELETE`. You will quickly learn that it's very easy to delete everything in Elasticsearch.

CHAPTER 4

Troubleshooting

4.1 I'm having problems using the config files!

We use `git-crypt` to encrypt our configuration files whilst allowing them to be versioned in git (meaning that we can also rollback configuration). To unlock the configuration encryption, you should install `git-crypt`, then run `bash install.sh` from the project root, and finally unlock the configuration using the key.

4.2 How do I restart the apache server after downtime?

```
sudo service httpd restart
```

4.3 How do I restart the luigi server after downtime?

```
sudo su - luigi
source activate py36
luigid --background --pidfile /var/run/luigi/luigi.pid --logdir /var/log/luigi
```

4.4 How do I perform initial setup to ensure the batchables will run?

- AWS CLI needs to be installed and configured:

```
pip install awscli
aws configure
```

AWS Access Key ID and Secret Access Key are set up in IAM > Users > Security Credentials Default region name should be `eu-west-1` to enable the error emails to be sent In AWS SES the sender and receiver email addresses need to be verified

- The config files need to be accessible and the PATH and LUIGI_CONFIG_PATH need to be amended accordingly

4.5 How can I send/receive emails from Luigi?

You should set the environmental variable `export LUIGI_EMAIL="<your.email@something>"` in your `.bashrc`. You can test this with `luigi TestNotificationsTask --local-scheduler --email-force-send`. Make sure your email address has been registered under AWS SES.

4.6 How do I add a new user to the server?

- add the user with `useradd --create-home username`
- add sudo privileges following these instructions
- add to ec2 user group with `sudo usermod -a -G ec2-user username`
- set a temp password with `passwd username`
- their home directory will be `/home/username/`
- copy `.bashrc` to their home directory
- create folder `.ssh` in their home directory
- copy `.ssh/authorized_keys` to the same folder in their home directory (DONT MOVE IT!!)
- `cd` to their home directory and perform the below
- chown their copy of `.ssh/authorized_keys` to their username: `chown username .ssh/authorized_keys`
- clone the nesta repo
- copy `core/config` files
- set password to be changed next login `chage -d 0 username`
- share the temp password and core pem file

If necessary: - `sudo chmod g+w /var/tmp/batch`

CHAPTER 5

Packages

Nesta's collection of tools for meaty tasks. Any processes that go into production come here first, but there are other good reasons for code to end up here.

CHAPTER 6

Production

Nesta's production system is based on Luigi pipelines, and are designed to be entirely run on AWS via the batch service. The main Luigi server runs on a persistent EC2 instance. Beyond the well documented Luigi code, the main features of the nesta production system are:

- `luigihacks.autobatch`, which facilates a managed `Luigi.Task` which is split, batched and combined in a single step. Currently only synchronous jobs are accepted. Asynchonous jobs (where downstream `Luigi.Task` jobs can be triggered) are a part of a longer term plan.
- `scripts.nesta_prepare_batch` which zips up the batchable with the specified environmental files and ships it to AWS S3.
- `scripts.nesta_docker_build` which builds a specified docker environment and ships it to AWS ECS.

6.1 License

Python Module Index

Index