ClarityNLP Documentation

Georgia Tech Research Institute

Sep 04, 2019

Contents

ClarityNLP is an "interoperable NLP" platform developed to streamline analysis of unstructured clinical text. The platform accelerates review of medical charts to extract data and identify patients for a wide variety of purposes, including research, clinical care, and quality metrics. ClarityNLP combines NLP techniques and libraries with a powerful query language, NLPQL, that lets you create and deploy NLP jobs quickly without a lot of custom configuration.

CHAPTER 1

Documentation

1.1 Setup

The instructions below will guide you through the ClarityNLP setup and installation process. There are several installation options for you to choose from:

1. **Local Machine Setup with Docker**

   Choose this option if you will be the only user of ClarityNLP, you want to install ClarityNLP on your laptop or desktop, and you want everything to be configured for you.

2. **Local Machine Setup without Docker**

   Choose this option if you will be the only user of ClarityNLP, you want to install ClarityNLP on your laptop or desktop, and you want to configure everything yourself.

3. **Server Setup**

   Choose this option if you anticipate supporting multiple users. This is a Docker-based installation with OAuth2 security.

1.1.1 Local Machine Setup

Local Machine Setup With Docker

The instructions below will get you up and running with a Docker-based ClarityNLP development environment on your laptop or desktop. We walk you through how to configure and deploy a set of Docker containers comprising a complete ClarityNLP installation for a single user. There is no need for you to separately install Solr, MongoDB, PostgreSQL, or any of the other technologies that ClarityNLP uses. Everything in the Docker containers has been setup and configured for you.

If you instead want to install ClarityNLP on a server to support multiple users, you should follow our *Server Setup* instructions.

If you want a single-user deployment without using Docker, then you need our *Local Machine Setup without Docker*.

Prerequisites

Download Source Code

```
git clone https://github.com/ClarityNLP/ClarityNLP
```

Initialize Submodules

```
cd ClarityNLP
git checkout <branch> # develop for latest, master for stable, or tagged version
git submodule update --init --recursive --remote
```

Install Docker

Follow the installation instructions.

These are the recommended Docker settings for ClarityNLP. In Docker, they can be updated via Docker > Preferences > Advanced.

- Memory: >8GB
- Disk: >256GB recommended, but can run on much less (depends on data needs)

Install Docker Compose

Follow the installation guide.

Install mkcert

The mkcert utility automatically creates and installs a local certificate authority (CA) in the system root store. It also generates locally-trusted certificates.

macOS

On macOS, use Homebrew.

```
brew install mkcert
brew install nss # if you use Firefox
```

or MacPorts.

```
sudo port selfupdate
sudo port install mkcert
sudo port install nss # if you use Firefox
```

Linux

On Linux, first install certutil.

```
sudo apt install libnss3-tools
    -or-
sudo yum install nss-tools
    -or-
sudo pacman -S nss
```

Then you can install using Linuxbrew

```
brew install mkcert
```

Windows

On Windows, use Chocolatey

```
choco install mkcert
```

or use Scoop

```
scoop bucket add extras
scoop install mkcert
```

Generate Development Certificates

First, create the local certificate authority:

```
mkcert -install
```

Run the following command at the root of the ClarityNLP project:

```
mkcert -cert-file certs/claritynlp.dev.crt -key-file certs/claritynlp.dev.key␣
↪claritynlp.dev "*.claritynlp.dev"
```

Extra Prerequisites for Windows

On Windows, install Cygwin and its dependencies

```
choco install cygwin
choco install cyg-get git git-completion make
```

Run the Stack

The first time running it will take some time to build the Docker images, but subsequent runs will occur quickly. First, start Docker if it is not already running. Next, open a terminal (Cygwin on Windows) at the project root and run the following for local development:

```
make start-clarity
```

The stack is running in the foreground, and can be stopped by simultaneously pressing the `CTRL` and `C` keys.

After stopping the stack, run this command to remove the containers and any networks that were created:

```
make stop-clarity
```

Tips & Tricks

To verify that the Docker containers are running, open a terminal and run:

```
docker ps
```

You should see a display that looks similar to this. There are 15 containers and all should have a status of `Up` when the system has fully initialized:

```
CONTAINER ID        IMAGE                                  COMMAND
↪CREATED             STATUS              PORTS                                      ↵
↪NAMES
55ac065604e5        claritynlp_ingest-api                  "/app/wait-for-it-ex..."
↪54 seconds ago      Up 24 seconds       1337/tcp                                   ↵
↪INGEST_API
ce2baf43bab0        claritynlp_nlp-api                     "/api/wait-for-it-ex..."
↪56 seconds ago      Up 54 seconds       5000/tcp                                   ↵
↪NLP_API
c028e60d1fab        redis:4.0.10                           "docker-entrypoint.s..."
↪About a minute ago  Up 56 seconds       6379/tcp                                   ↵
↪REDIS
4e1752025734        jpillora/dnsmasq                       "webproc --config /e..."
↪About a minute ago  Up 56 seconds       0.0.0.0:53->53/udp                         ↵
↪DNSMASQ
2cf1dd63257a        mongo                                  "docker-entrypoint.s..."
↪About a minute ago  Up 55 seconds       27017/tcp                                  ↵
↪NLP_MONGO
34385b8f4306        claritynlp_nlp-postgres                "docker-entrypoint.s..."
↪About a minute ago  Up 56 seconds       5432/tcp                                   ↵
↪NLP_POSTGRES
500b36b387b7        claritynlp_ingest-client               "/bin/bash /app/run...."
↪About a minute ago  Up 56 seconds       3000/tcp, 35729/tcp                        ↵
↪INGEST_CLIENT
f528b68a7490        claritynlp_dashboard-client            "/bin/bash /app/run...."
↪About a minute ago  Up 56 seconds       3000/tcp, 35729/tcp                        ↵
↪DASHBOARD_CLIENT
8290a3846ae0        claritynlp_results-client              "/bin/bash /app/run...."
↪About a minute ago  Up 56 seconds       3000/tcp, 35729/tcp                        ↵
↪RESULTS_CLIENT
77fce3ae48fc        claritynlp_identity-and-access-proxy   "pm2-dev process.json"
↪About a minute ago  Up 57 seconds       6010/tcp                                   ↵
↪IDENTITY_AND_ACCESS_PROXY
b6610c74ec4c        claritynlp_nlp-solr                    "docker-entrypoint.s..."
↪About a minute ago  Up 56 seconds       8983/tcp                                   ↵
↪NLP_SOLR
45503f0fd389        claritynlp_identity-provider           "docker-entrypoint.s..."
↪About a minute ago  Up 57 seconds       5000/tcp                                   ↵
↪IDENTITY_PROVIDER
```

(continues on next page)

(continued from previous page)

```
6dc0f7f21a48        claritynlp_nginx-proxy                  "/app/docker-entrypo..."  ␣
↪About a minute ago   Up 56 seconds       0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp  ␣
↪NGINX_PROXY
1d601b064a1c        axiom/docker-luigi:2.7.1                "/sbin/my_init --qui..."  ␣
↪About a minute ago   Up 57 seconds       8082/tcp                                  ␣
↪LUIGI_SCHEDULER
7ab4b8e19c86        mongo:3.4.2                             "docker-entrypoint.s..."  ␣
↪About a minute ago   Up 58 seconds       27017/tcp                                 ␣
↪INGEST_MONGO
```

The Luigi container will monitor for active tasks. Once everything initializes, you should periodically see the following lines in the console output:

```
LUIGI_SCHEDULER    | 2018-10-16 19:46:19,149 luigi.scheduler INFO     Starting pruning␣
↪of task graph
LUIGI_SCHEDULER    | 2018-10-16 19:46:19,149 luigi.scheduler INFO     Done pruning␣
↪task graph
```

ClarityNLP Links

The user interface (UI) components of ClarityNLP can be accessed on your machine by opening a web browser and entering the URLs provided below. Each different user interface component has been mapped to a unique URL in the `.dev` top level domain.

All Docker containers must be fully initialized for the UI components to become active.

Dashboard

The ui_dashboard is the main user interface to ClarityNLP. It provides controls for ingesting documents, creating NLPQL files, accessing results and lots more.

Dashboard URL: https://dashboard.claritynlp.dev

Solr Administrative User Interface

Solr provides an administrative user interface that you can use to configure and explore your ClarityNLP Solr instance. The Apache project provides full documentation on the admin UI which you can find here.

Perhaps the most useful component of this UI is the query tool, which lets you submit queries to Solr and find documents of interest. The ClarityNLP Solr installation provides more than 7000 documents in a core called `sample`.

Solr Admin Interface URL: https://solr.claritynlp.dev

Luigi Task Monitor

The Luigi project provides a task monitor that displays information on the currently running ClarityNLP job. ClarityNLP processes documents by dividing the workload into parallel tasks that are scheduled by Luigi. The task monitor displays the number of running tasks, how many have finished, any failures, etc. You can update the task counts by simply refreshing the page.

Lugi Task Monitor URL: https://luigi.claritynlp.dev

Ingest Client

The *Ingest Client* provides an easy-to-use interface to help you load new documents into your ClarityNLP Solr instance. It also helps you map the fields in your documents to the fields that ClarityNLP expects.

Ingest Client URL: https://ingest.claritynlp.dev

Results Viewer

The *Results Viewer* helps you examine the results from each of your ClarityNLP runs. It highlights specific terms and values and provides an evaluation mechanism that you can use to score the results that ClarityNLP found.

Clarity Results Viewer URL: https://viewer.claritynlp.dev

NLP API

<TODO - example of how to POST an NLPQL file using Postman or curl with access tokens>

Local Machine Setup without Docker

This page provides instructions on how to run ClarityNLP locally on your machine **without** having to use Docker or OAuth2. We call this a *native* installation of ClarityNLP. It is much simpler to use Docker, since everything is provided and configured for you. But if you want more control over your ClarityNLP installation and you prefer to configure everything yourself, then these are the instructions you need.

This installation is also useful if you neither need nor want the OAuth2 security layers built into the Docker version of ClarityNLP. A native installation is emphatically **NOT** appropriate for patient data that must be protected in a HIPAA-compliant manner. So only store de-identified public data in your Solr instance if you choose to do this.

Overview

There are five major components in a ClarityNLP installation: Solr, PostgreSQL, MongoDB, Luigi, and Flask.

ClarityNLP uses Solr to index, store, and search documents; Postgres to store job control data and lots of medical vocabulary; Mongo to store results; Luigi to control and schedule the various processing tasks, and Flask to provide API endpoints and the underlying web server.

A native installation means that, at a minimum, Luigi and Flask are installed and run locally on your system. Solr, Postgres, and Mongo can also be installed and run locally on your system, or one or more of these can be hosted elsewhere.

A university research group, for example, could have a hosted Solr instance on a VPN that is accessible to all members of the group. The Solr instance might contain MIMIC or other de-identified, public data. Members of the research group running a native ClarityNLP installation would configure their laptops to use the hosted Solr instance. This can be accomplished via settings in a ClarityNLP configuration file, as explained below. These users would install and run Postgres, Mongo, Luigi, and Flask on their laptops.

At GA Tech we have hosted versions of Solr, Postgres, and MongoDB. Our native ClarityNLP users only need to install and run Luigi and Flask on their laptops, and then setup their configuration file to "point" to the hosted instances.

These flexible configuration options are also available with the container-based, secure version of ClarityNLP.

The instructions below have been tested on:

- MacOS 10.14 "Mojave"

- MacOS 10.13 “High Sierra”
- Ubuntu Linux 18.04 LTS “Bionic Beaver”

Recent versions of MongoDB, PostgreSQL, and Solr are assumed:

- MongoDB version 3.6 or greater
- PostgreSQL version 10 or 11
- Solr version 7 or 8

Roadmap

This installation and configuration process is somewhat lengthy, so here’s a high-level overview of what we’ll be doing.

First, we’ll need to setup and install the source code, the necessary python libraries, and all of the associated python and non-python dependencies. We will perform the installation inside of a custom conda-managed environment so that ClarityNLP will not interfere with other software on your system.

Next we’ll install and/or configure Solr, PostgreSQL, and MongoDB, depending on whether you have access to hosted instances or not.

Then we’ll ingest some test documents into Solr and run a sample NLPQL file so that we can verify that the system works as expected.

After that we’ll show you where you can find instructions for ingesting your own documents into Solr, after which you will be ready to do your own investigations.

The instructions below denote MacOS-specific instructions with **[MacOS]**, Ubuntu-specific instructions with **[Ubuntu]**, and instructions valid for all operating systems with **[All]**.

Install the Prerequisites

[MacOS] Install the Homebrew package manager by following the instructions provided at the Homebrew website. We prefer to use Homebrew since it allows packages to be installed and uninstalled without superuser privileges.

After installing homebrew, open a terminal window and update your homebrew installation with:

```
brew update
brew upgrade
```

Next, use homebrew to install the `git` version control system, the `curl` command line data transfer tool, and the `wget` file transfer tool with these commands:

```
brew install git curl wget
```

[Ubuntu] Update your system using the `apt` package manager with:

```
sudo apt update
sudo apt upgrade
```

Then use `apt` to install the three tools:

```
sudo apt install git curl wget
```

[All] Solr requires the java runtime to be installed on your system. In a terminal window run this command:

```
java --version
```

If you see a message about the command `java` not being found, then you need to install the java runtime. Please visit the Oracle Java download site and follow the instructions to download and install the latest version of the Java runtime environment (JRE).

Next, visit the Conda website and install either the Anaconda python distribution or its much smaller Miniconda cousin. Anaconda provides a full python-based numerical computing and machine learning stack. Miniconda provides a minimal python installation. Both give you the `conda` package manager, an essential tool for resolving labyrinthine dependencies among python and non-python packages. The installation package and instructions for both are provided at the Anaconda website. For these instructions we will assume that you choose the smaller Miniconda distribution.

Important: download the Miniconda installation package for the latest python 3 release, not python 2.7.

After installing Miniconda, update to the latest version of `conda` with:

```
conda update -n base -c defaults conda
```

Clone the ClarityNLP GitHub Repository

Open a terminal window on your system and change directories to wherever you want to install ClarityNLP. Create a new folder called `ClarityNLPNative`, to emphasize that it will hold a version of ClarityNLP configured for running locally on your system without Docker or OAuth2. You can create this folder, clone the repo, and initialize all submodules with these commands:

```
cd /some/location/on/your/disk
mkdir ClarityNLPNative
cd ClarityNLPNative
git clone --recurse-submodules https://github.com/ClarityNLP/ClarityNLP.git
cd ClarityNLP
```

This command sequence will give you an up-to-date checkout of the master branch of the main ClarityNLP project. It will also checkout the latest master branch of all git submodules (additional code that ClarityNLP needs).

The master branch of the git repository holds the most stable and well-tested version of ClarityNLP. If you instead want the latest development code, with the caveat that it will be less mature than the code in the master branch, checkout the `develop` branch of the repo with these additional commands:

```
git checkout develop
git submodule foreach git pull origin develop
```

After checking out your desired branch of the repository, change to the `native_setup` folder of the repo with:

```
cd native_setup
```

Create the Conda Environment for ClarityNLP

From the `ClarityNLPNative/ClarityNLP/native_setup` folder, create a new conda managed environment with:

```
conda create --name claritynlp python=3.6
conda activate claritynlp
conda config --env --append channels conda-forge
```

(continues on next page)

(continued from previous page)

```
conda install --file conda_requirements.txt
pip install -r conda_pip_requirements.txt
```

The conda version of `pip` knows about conda environments and will install the packages listed in `conda_pip_requirements.txt` into the claritynlp custom environment, NOT the system folders.

You can activate the claritynlp custom environment with the command

```
conda activate claritynlp
```

Whenever the claritynlp environment is active, the command line in the terminal window displays `(claritynlp)` to the left of the prompt. If the default environment is active it will display `(base)` instead.

Always activate the claritynlp environment whenever you want to do anything with ClarityNLP from a terminal window.

Install Additional Model Files

ClarityNLP uses the spacy and nltk natural language processing libraries, which require additional support files. From the same terminal window in the `native_setup` folder, run these commands to install the support files:

```
conda activate claritynlp   # if not already active
python -m spacy download en_core_web_sm
python ../nlp/install_models.py
```

Setup MongoDB

ClarityNLP stores results in MongoDB. If you do not have access to a hosted MongoDB installation, you will need to install it on your system.

[MacOS] Use Homebrew to install MongoDB with:

```
brew install mongodb
```

After the installation finishes, run the command `brew info mongodb`, which displays information about how to start the MongoDB server. You can either configure the server to start automatically each time your system reboots, or you can start the server manually. We will assume manual startup, which can be accomplished by opening another terminal window and running this command (assumes the default path to the mongo config file):

```
mongod --config /usr/local/etc/mongod.conf
```

After the server initializes it will deactivate the prompt in the terminal window, indicating that it is running.

[Ubuntu] Use `apt` to install MongoDB with:

```
sudo apt install mongodb
```

The installation process on Ubuntu should automatically start the MongoDB server. Verify that it is active with:

```
sudo systemctl status mongodb
```

You should see a message stating that the `mongodb.service` is active and running. If it is not, start it with:

```
sudo systemctl start mongodb
```

Then repeat the status check to verify that it is running.

[All] Now start up the Mongo **client** and find out if it can communicate with the running MongoDB server. From a terminal window start the MongoDB client by running `mongo`. If the client launches successfully you should see a > prompt. Enter `show databases` at the prompt and press enter. The system should respond with at least the *admin* database. If you see this your installation should be OK. You can stop the client by typing `exit` at the prompt.

If you have access to a hosted MongoDB instance, you will need to know the hostname for your `mongod` server as well as the port number that it listens on. If your hosted instance requires user accounts, you will also need to know your username and password. These will be entered into the `project.cfg` file in a later step below.

Setup PostgreSQL

Now we need to install and configure PostgreSQL. ClarityNLP uses Postgres for job control and for storing OMOP vocabulary and concept data.

[MacOS] Perhaps the easiest option for installing Postgres on MacOSX is to download and install Postgres.app, which takes care of most of the setup and configuration for you. If you do not have access to a hosted Postgres server, download the .dmg file from the Postgres.app website, run the installer, and click *initialize* to create a new server.

After everything is installed and running, you will see an elephant icon in the menu bar at the upper right corner of your screen. Click the icon and a menu will appear. The button in the lower right corner of the menu can be used to start and stop the database server. For now, click the button and stop the server, since we need to make a small change to the postgres configuration file.

[Ubuntu] Install postgres with:

```
sudo apt install postgresql
```

The installation process should automatically start the postgres server, as it did with the MongoDB installation. For now, stop the server with:

```
sudo systemctl stop postgresql
```

Edit the PostgreSQL Config File

You will need to follow these configuration steps as well if you have a hosted Postgres instance. You may need to ask your local database admin to perform the configuration, depending on whether or not you have superuser privileges for your particular installation. The location of the data directory on your hosted instance will likely differ from that provided below, which is specific to a local installation.

[MacOS] With the Postgres server stopped, click the elephant icon, click the `Open Postgres` menu item, and then click the `Server Settings` button on the dialog that appears. Note the location of the data directory, which defaults to `~/Library/Application Support/Postgres/var-11`. The `postgresql.conf` file is located in the data directory and contains various important parameters that govern the operation of the database. We need to edit one of those params to make the data ingest process run more smoothly.

[Ubuntu] The postgres config file for Postgres 10 is stored by default in `/etc/postgresql/10/main/postgresql.conf`. If you installed Postgres 11 the 10 should be replaced by an 11. This file is owned by the special `postgres` user. To edit the file, switch to this user account with:

```
sudo -i -u postgres
whoami
```

The `whoami` command should display `postgres`.

[All] Open a text editor, browse to the location indicated above and open the file `postgresql.conf`. Search the file for the entry `max_wal_size`, which governs the size of the write-ahead log (hence the WAL acronym). If the entry happens to be commented out, uncomment it. Set its value to 30GB (if the value is already greater than 30GB don't change it). By doing this we prevent checkpoints from occurring too frequently and slowing down the data ingest process. Save the file after editing.

[Ubuntu] Log out as the `postgres` user with:

```
exit
```

Then restart the Postgres server with either:

[MacOS] Click on the elephant icon and press the start button.

[Ubuntu] Use `systemctl` to start it:

```
sudo systemctl start postgresql
```

Create the Database and a User Account

With the database server installed, configured, and running, we now need to create a user account. Open a terminal and browse to `ClarityNLPNative/ClarityNLP/utilities/nlp-postgres`. From this folder run the command appropriate to your operating system to start `psql`:

[MacOS]

```
psql postgres
```

[Ubuntu]

```
sudo -u postgres psql
```

Then run this command sequence (we suggest using a better password) to setup the database:

```
CREATE USER clarity_user WITH LOGIN PASSWORD 'password';
CREATE DATABASE clarity;
\connect clarity
\i ddl/ddl.sql
\i ddl/omop_vocab.sql
\i ddl/omop_indexes.sql
GRANT USAGE ON SCHEMA nlp TO clarity_user;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA nlp TO clarity_user;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA nlp TO clarity_user;
```

These commands create the database, setup the tables and indexes, and grant the `clarity_user` sufficient privileges to use it with ClarityNLP.

Load OMOP Vocabulary Files

THIS STEP IS OPTIONAL. The OMOP vocabulary and concept data is used by the ClarityNLP synonym expansion macros. Synonym expansion is an optional feature of ClarityNLP. If you are unfamiliar with OMOP or do not forsee a need for such synonym expansion you can safely skip this step. The ingestion process is time-consuming and could take from one to two hours or more, depending on the speed of your system. If you only want to explore basic features of ClarityNLP you do not need to load this data, and you can skip ahead to the Solr setup instructions.

If you do choose to load the data, then keep your `psql` terminal window open. **From a different terminal window** follow these steps to download and prepare the data for ingest:

```
cd /tmp
mkdir vocabs
cd vocabs
wget http://healthnlp.gtri.gatech.edu/clarity-files/omop_vocabulary_set.zip
unzip omop_vocabulary_set.zip
rm omop_vocabulary_set.zip
```

You should see these files in `/tmp/vocabs` after unzipping:

```
DOMAIN.csv
CONCEPT_CLASS.csv
CONCEPT.csv
CONCEPT_ANCESTOR.csv
RELATIONSHIP.csv
CONCEPT_SYNONYM.csv
VOCABULARY.csv
CONCEPT_RELATIONSHIP.csv
DRUG_STRENGTH.csv
```

Go back to your `psql` window and begin the process of loading data into the database with:

```
\i dml/copy_vocab.sql
```

As mentioned above, the loading process could take a **long** time, possibly more than two hours, depending on the speed of your system. As the load progresses, it should gradually generate the following output:

```
SET
COPY 2465049
COPY 2781581
COPY 23396378
COPY 21912712
COPY 3878286
COPY 27
COPY 446
COPY 321
COPY 40
```

Once you start the loading process, just let it run. . . it will eventually finish. After loading completes, log out with the command `\q`. You can close this window and the `tmp/vocabs` window.

Setup Solr

ClarityNLP uses Solr as its document store. If you do not have access to a hosted Solr instance you will need to install it on your system.

[MacOS] Use Homebrew to install Solr with:

```
brew install solr
```

When the installation finishes run the command `brew info solr` to learn how to start Solr. You can either have it start on boot or on demand with the command

```
solr start
```

Start the solr server.

[Ubuntu] Ubuntu does not seem to provide a suitable apt package for Solr, so you will need to download the Solr distribution from the Apache web site. Open a web browser to the Solr download site and download the binary release for the latest version of Solr 8. For now we will assume that you download the 8.1.1 **binary** release, which is in the file `solr-8.1.1.tgz`.

Open a terminal window and run these commands to unzip the distribution into your home directory:

```
cd ~
mkdir solr
tar -C solr -zxvf ~/Downloads/solr-8.1.1.tgz
mv ~/solr/solr-8.1.1 ~/solr/8.1.1
```

Open a text editor and add this line to your `.bashrc` file, which places the Solr binaries on your path:

```
export PATH=~/solr/8.1.1/bin:$PATH
```

Close the text editor, exit the terminal window, and open a new terminal window to update your path. Run `which solr` and verify that `~/solr/8.1.1/bin/solr` is found.

Start your Solr server by running:

```
solr start
```

[All] After starting Solr, check to see that it is running by opening a web browser to `http://localhost:8983` (or the appropriate URL for your hosted instance). You should see the Solr admin dashboard. If you do, your Solr installation is up and running.

We need to do some additional configuration of the Solr server and ingest some test documents. We provide a python script to do this for you. **This script assumes that you are running a recent version of Solr, version 7 or later.** If you are running an older version this script **will not work**, since some field type names changed at the transition from Solr 6 to Solr 7.

Open a terminal window to `ClarityNLPNative/ClarityNLP/native_setup`. If you installed Solr on your local system run:

```
conda activate claritynlp
python ./configure_solr.py
```

If you use a hosted Solr instance, you should run these commands instead, replacing the `<hostname>` and `<port_number>` placeholders with the values for your hosted instance:

```
conda activate claritynlp
python ./configure_solr.py --hostname <hostname_string> --port <port_number>
```

This script creates a Solr core named `claritynlp_test`, adds some custom fields and types, and loads test documents contained in four `.csv` files. You should confirm that the files `sample.csv`, `sample2.csv`, `sample3.csv`, and `sample4.csv` were loaded successfully (load statements appear in the console as the script runs). If the load failed for any reason an error message will be written to stdout.

If the script ran without error, your `claritynlp_test` Solr core should have ingested 7016 documents. Verify this by opening a web browser to `http://localhost:8983`, or if you have a hosted Solr instance, to its admin page. From the core selector at the left of the screen, select the `claritynlp_test` core and look in the `Statistics` window. The value of the `Num Docs` field should equal 7016.

ClarityNLP expects the ingested documents to have a minimal set of fields, which appear in the next table:

Field Name	Description
id	a unique ID for this document
report_id	a unique ID for this document (can use same value as `id` field)
source	the name of the document set, the name of your institution, etc.
subject	a patient ID, drug name, or other identifier
report_type	type of data in the document, i.e. `discharge summary`, `radiology`, etc.
report_date	timestamp in a format accepted by Solr: • `YYYY-MM-DDThh:mm:ssZ` • `YYYY-MM-DDThh:mm:ss.fZ` • `YYYY-MM-DDThh:mm:ss.ffZ` • `YYYY-MM-DDThh:mm:ss.fffZ`
report_text	the actual text of the document, plain text

The test documents have all been configured with these fields. If you decide to ingest additional documents into the `claritynlp_test` Solr core, you will need to ensure that they contain these fields as well. Additional information on document ingestion can be found here.

Python scripts for ingesting some common document types can be found here.

Setup the Project Configuration File

In the `ClarityNLPNative/native_setup` directory you will find a file named `project.cfg`. This file gets loaded on startup and it configures Clarity to run locally on your system.

If you plan to use hosted instances of either Solr, Postgres, or MongoDB, you will need to edit the file and set the values appropriate for your system. The file has a simple `key=value` format for each parameter. The Solr parameters are located under the `[solr]` header, the Postgres params under the `[pg]` header, and the MongoDB params under the `[mongo]` header.

For instance, if you installed everything locally, but you changed the PostgreSQL password above when you created the user account, you need to open `project.cfg` in a text editor, locate the `[pg]` section, find the `password=password` entry, and change the text on the right side of the equals sign to the password that you used. If you used a password of `jx8#$04!Q%`, change the password line to `password=jx8#$04!Q%`.

Make the appropriate changes for Solr, Postgres, and MongoDB to conform to your desired configuration. Note that the username and password entries for MongodB are commented out. It is possible to use MongoDB without having to create a user account. If this is the case for your system, just leave these entries commented out. Otherwise, uncomment them and set the values appropriate for your system.

If you followed the instructions above *exactly* and installed everything locally, you do not need to change anything in this file.

The provided `project.cfg` file tells ClarityNLP to use `/tmp` as the location for the log file and various temporary files needed during the run. If you want to put these files somewhere else, create the desired folders on your system, make them writable, and set the paths in the `[tmp]` and `[log]` sections of `project.cfg`. The paths would look like this after any changes:

```
[tmp]
dir=/path/to/my/preferred/tmp/dir
```

(continues on next page)

(continued from previous page)

```
[log]
dir=/path/to/my/preferred/log/dir
```

Double-check all entries in this file! You will have problems getting the system to run if you have typos or other errors for these parameters.

Once you are satisifed that the data in the file is correct, copy `project.cfg` from the `native_setup` folder into the `nlp` folder, which is where ClarityNLP expects to find it:

```
cp project.cfg ../nlp/project.cfg
```

Running Locally without Docker

Now we're finally ready to run. Here are the instructions for running a job with your native ClarityNLP system. We open several terminal windows to start the various servers and schedulers. You can reduce the number of windows by configuring Mongo, Postgres, and Solr to start as background processes after each reboot, as mentioned above.

1. Start Solr

If you installed Solr locally and chose the manual start method, start Solr by opening a terminal window and running `solr start`.

Verify that you can communicate with your Solr core by pinging it. For a local installation, open a Web browser and visit this URL: `http://localhost:8983/solr/claritynlp_test/admin/ping`. For a hosted instance, change `localhost` to whatever is appropriate for your system.

The Web browser should display a status of `OK` in the final line of output if it is connected. If you get an HTTP 404 error, make recheck your URL and make sure that your Solr instance is actually running.

2. Start the MongoDB Server

If you installed MongoDB locally, launch the the `mongod` server with one of these options:

[MacOS] Provide the path to your local MongoDB config file as follows (this command uses the default location):

```
mongod --config /usr/local/etc/mongod.conf
```

[Ubuntu]

```
sudo systemctl start mongodb
```

Verify that the mongo server is running by typing `mongo` into a terminal to start the mongo client. It should connect to the database and prompt for input. Exit the client by typing `exit` in the terminal.

For a hosted MongoDB instance you need to supply the connection params from the terminal. If your Mongo installation does not require accounts and passwords, connect to it with this command, replacing the `<hostname or ip>` and `<port number>` placeholders with values appropriate for your system:

```
mongo --host <hostname or ip> --port <port number>
```

If your hosted instance requires a user name and password, you will need to supply those as well. More info on connecting to a remote Mongo server can be found here.

3. Start the Postgres Server

If you installed Postgres locally:

[MacOS] Start the server by clicking the elephant icon in the menu bar at the upper right corner of your screen. Press the start button at the lower right of the popup menu.

[Ubuntu] Start the server with:

```
sudo systemctl start postgresql
```

Verify that your server is available by running the command `pg_isready` from a terminal window. It should report `accepting connections`.

If you use a hosted Postgres instance, check to see that it is up and running with this command, replacing the hostname and port number with values suitable for your installation:

```
pg_isready -h <hostname> -p <port number>
```

If your Postgres server is running it should respond with `accepting connections`.

4. Start the Luigi Task Scheduler

ClarityNLP uses Luigi to schedule and manage the data processing tasks. Luigi must be started manually in a native setup.

We will run Luigi from a dedicated directory, `~/tmp/luigi`. Open another terminal window and create `~/tmp/luigi` with these commands (this only needs to be done once):

```
mkdir -p ~/tmp/luigi
cd ~/tmp/luigi
mkdir logs
```

Launch Luigi with:

```
conda activate claritynlp
cd ~/tmp/luigi
luigid --pidfile pid --logdir logs --state-path statefile
```

Luigi should start and the command prompt should become inactive. Keep Luigi running for your entire ClarityNLP session. You only need to start Luigi once, even if you plan to run multiple ClarityNLP jobs.

5. Start the Flask Web Server

ClarityNLP uses Flask as the underlying web framework. Flask must be started manually in a native setup.

Open yet another terminal window, cd to the `ClarityNLPNative/ClarityNLP/nlp` directory, and launch the web server with:

```
conda activate claritynlp
export FLASK_APP=api.py
python -m flask run
```

Just like Luigi, the Flask web server only needs to be started once. The web server prints startup information to the screen as it initializes. You can safely ignore any `No section:` warnings. When initialization completes you should see output similar to this:

```
* Serving Flask app "nlp.api"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

At this point ClarityNLP is fully initialized and waiting for commands.

6. Run a Validation Job

Open (yet another) terminal window and cd to `ClarityNLPNative/ClarityNLP/native_setup`. Run the `ls` command and note the file `validation0.nlpql`. This is an NLPQL file that runs several ClarityNLP tasks on a special validation document that was loaded into the `claritynlp_test` Solr core during setup.

When we run this validation job, ClarityNLP will process the validation document, run the validation tasks, and write results to MongoDB. We can extract the results into a CSV file for easy viewing and then run a special python script to check that the results are correct.

You launch a ClarityNLP job by performing an HTTP POST of your NLPQL file to the ClarityNLP `nlpql` API endpoint. Since the local running instance of ClarityNLP is listening at `http://localhost:5000`, the appropriate URL is `http://localhost:5000/nlpql`. We will see how to post the file using the `curl` command line tool below. If you are familiar with Postman or other HTTP clients you could certainly use those instead of `curl`. Any HTTP client that can POST files as plain text should be OK.

Before running the NLPQL file, we should first check it for syntax errors. That can be accomplished by POSTing the NLPQL file to the `nlpql_tester` API endpoint. From your terminal window run these commands to do so:

```
conda activate claritynlp
curl -i -X POST http://localhost:5000/nlpql_tester -H "Content-type:text/plain" --
↪data-binary "@validation0.nlpql"
```

The curl command should generate output that looks similar to this:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 2379
Access-Control-Allow-Origin: *
Server: Werkzeug/0.15.2 Python/3.6.6
Date: Thu, 06 Jun 2019 00:37:26 GMT

{
    "owner": "claritynlp",
     "name": "Validation 0",
     "population": "All",
     "context": "Patient",

     <lots of content omitted...>

     "debug": false,
     "limit": 100,
     "phenotype_id": 1
}
```

This is the JSON representation of the NLPQL file generated by the ClarityNLP front end. If you see JSON output similar to this your syntax is correct. If you do not get JSON output then something is wrong with your NLPQL syntax. There should be an error message printed in the Flask window. The `validation0.nlpql` file has been checked and should contain no syntax errors.

After the syntax check we're ready to run the job. POST the NLPQL file to the `nlpql` endpoint with this command:

```
curl -i -X POST http://localhost:5000/nlpql -H "Content-type:text/plain" --data-
↪binary "@validation0.nlpql"
```

The system should accept the job and print out a message stating where you can download the results. The message should look similar to this:

```
{
    "job_id": "1",
    "phenotype_id": "1",
    "phenotype_config": "http://localhost:5000/phenotype_id/1",
    "pipeline_ids": [
        1
    ],
    "pipeline_configs": [
        "http://localhost:5000/pipeline_id/1"
    ],
    "status_endpoint": "http://localhost:5000/status/1",
    "results_viewer": "?job=1",
    "luigi_task_monitoring": "http://localhost:8082/static/visualiser/index.html
↪#search__search=job=1",
    "intermediate_results_csv": "http://localhost:5000/job_results/1/phenotype_
↪intermediate",
    "main_results_csv": "http://localhost:5000/job_results/1/phenotype"
 }
```

The `job_id` increments each time you submit a new job. The system should launch approximately 22 tasks to run the commands in this sample file. If you open a web browser to the `luigi_task_monitoring` URL, you can watch the tasks run to completion in the luigi task status display. Just refresh the window periodically to update the task counts.

After the job finishes you can download a CSV file to see what ClarityNLP found. The `intermediate_results_csv` file contains all of the raw data values that the various tasks found.

To check the results, you need to generate a CSV file from the intermediate data with a comma for the record delimiter, **not a tab**. A tab character seems to be the default delimiter for Microsoft Excel.

Excel users can correct this as follows. Assuming that you have the intermediate result file open in Excel, press the key combination <COMMAND>-A. This should highlight the leftmost column of data in the spreadsheet. After highlighting, click the `Data` menu item, then press the `Text to Columns` icon in the ribbon at the top. When the wizard dialog appears, make sure the `Delimited` radio button is highlighted. Click `Next`. For the delimters, make sure that `Comma` is checked and that `Tab` is unchecked. Then click the `Finish` button. The data should appear neatly arranged into columns. Then click the `File|Save As...` menu item. On the dialog that appears, set the `File Format` combo box selection to `Comma Separated Values (.csv)`. Make sure that a `.csv` extension appears in the `Save As` edit control at the top of the dialog. Give the file a new name if you want (but with a `.csv` extension), then click the `Save` button.

Users of other spreadsheet software will need to consult the documentation on how to save CSV files with a comma for the record separator.

With the file saved to disk in proper CSV format, run this command from the `ClarityNLPNative/ClarityNLP/native_setup` folder to check the values:

```
conda activate claritynlp  # if not already active
python ./validate_results0.py --file /path/to/your/csv/file.csv
```

This command runs a python script to check each result. If the script finds no errors it will print `All results are valid.` to stdout. If ClarityNLP is working properly no errors should be found.

Shutdown

Perform these actions to completely shutdown ClarityNLP on your system:

1. Stop the Flask webserver by entering <CTRL>-C in the flask terminal window.
2. Stop the Luigi task scheduler by entering <CTRL>-C in the luigi terminal window.
3. MacOS users can stop the MongoDB database server by entering <CTRL>-C in the MongoDB terminal window. Ubuntu users can run the command `sudo systemctl stop mongodb`.
4. Stop Solr by entering `solr stop -all` in a terminal window.
5. MacOS users can stop Postgres by first clicking on the elephant icon in the menu bar at the upper right corner of the screen. Click the stop button on the menu that appears. Ubuntu users can run the command `sudo systemctl stop postgresql`.

Alternatively, you could just terminate Flask and Luigi and keep the other servers running if you plan to run more jobs later.

If you restart, always start Luigi **before** Flask, exactly as documented above.

Final Words

An introduction to NLPQL can be found here.

Additional information on how to run jobs with ClarityNLP can be found in our Cooking with Clarity sessions. These are Jupyter notebooks presented in a tutorial format. Simply click on any of the `.ipynb` files to open the notebook in a Web browser. These notebooks provide in-depth explorations of topics relevant to computational phenotyping.

1.1.2 Server Setup

Server Setup

Nonsecure Deployment

Docker Setup

1. Install both Docker and Docker-Compose on your machine. Go here to install Docker, find your OS and follow instructions. Go here to install Docker Compose.
2. Run `git clone https://github.com/ClarityNLP/ClarityNLP [folder-name]`
3. Initialize submodules `git submodule update --init --recursive`
4. Add .env file, use .env.example as a start:

```
  cd [folder-name]
touch .env
cat .env.example >> .env
```

1. Build images and run containers `docker-compose -f docker-compose.prod.yml up --build -d`

Secure Deployment

Updating to download latest changes

From the command line, run:

```
git pull
git submodule update --recursive
```

1.1.3 Document Ingestion

General Document Ingestion

See guide to Solr for more information about Solr setup with ClarityNLP.

Solr has built-in APIs for ingesting documents, which are documented here. The simplest way is generally to use `curl` to upload JSON, CSV, or XML. Documents need to be pre-processed as plain text before they are uploaded into ClarityNLP.

Sample JSON upload for ClarityNLP:

```
curl -X POST -H 'Content-Type: application/json' 'http://localhost:8983/solr/report_
↪core/update/json/docs' --data-binary '
 {
        "report_type":"Report Type",
        "id":"1",
        "report_id":"1",
        "source":"My Institution",
        "report_date":"1970-01-01T00:00:00Z",
        "subject":"the_patient_id_or_other_identifier",
        "report_text":"Report text here"
    }'
```

Upload Scripts

A collection of scripts for ingesting popular datasets (MIMIC, AACT Clinical Trials, Gleason Pathology Documents, etc.) is available here.

1.2 User Guide

1.2.1 Overview

ClarityNLP is is a natural language processing platform designed to accelerate review of medical charts for a wide variety of purposes, including research, clinical care, quality metrics, and risk adjustment. This guide is intended for end users of ClarityNLP.

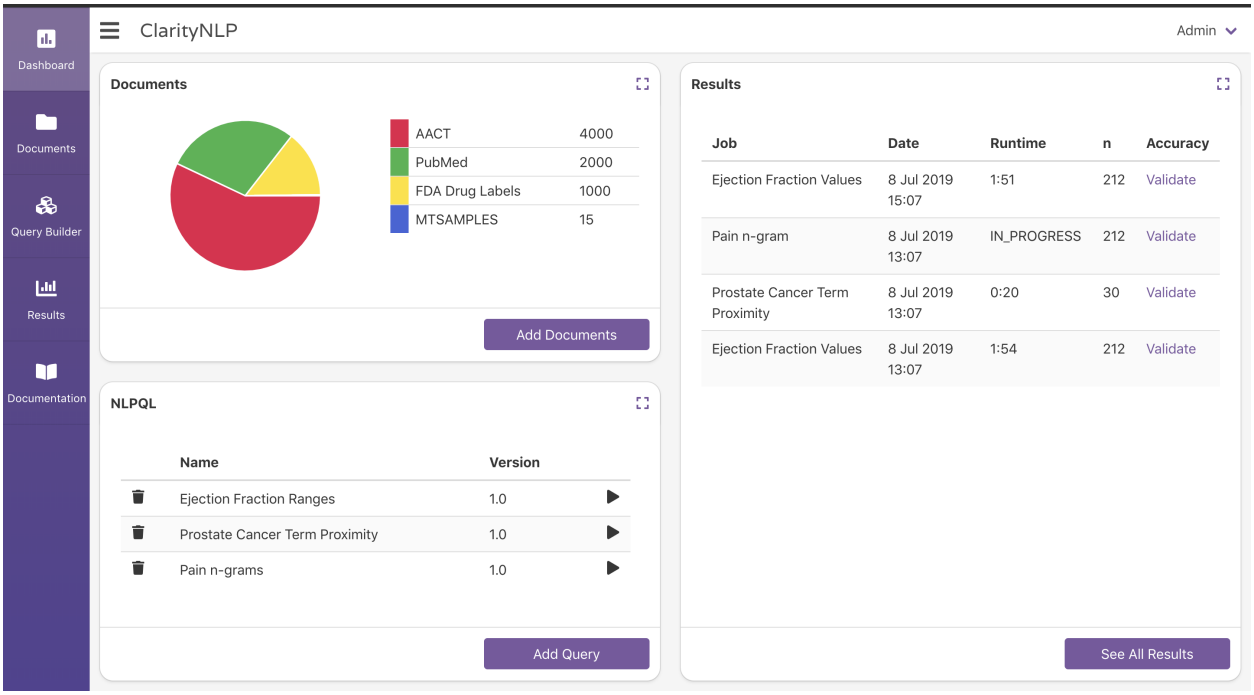
We host webinars biweekly on Wednesday where we present and discuss topics around ClarityNLP. See upcoming and previous sessions here. For more information, contact us.

1.2.2 User Interface

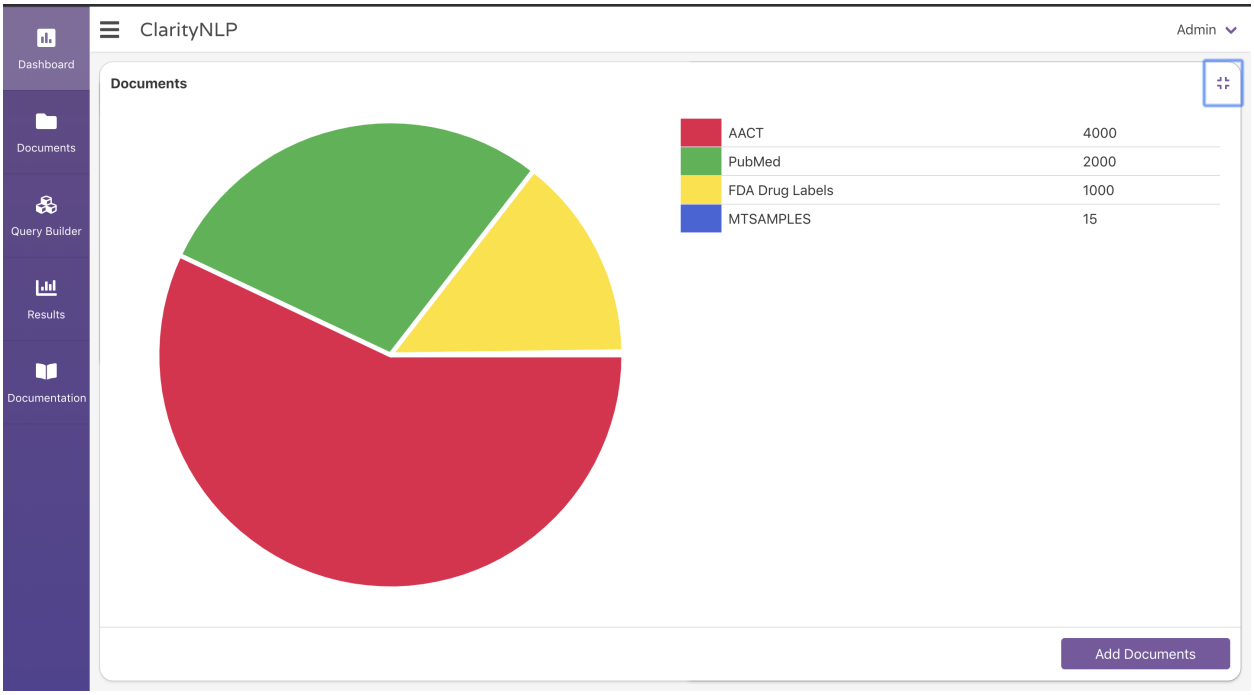
Dashboard

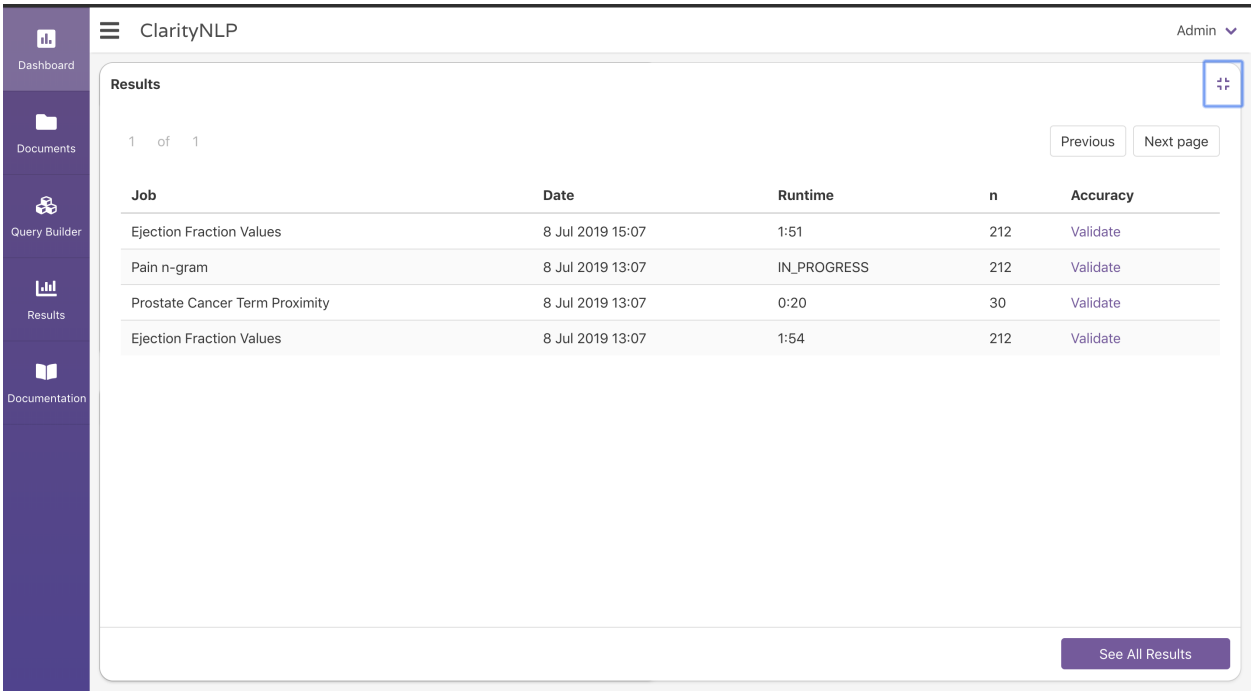
The Dashboard is designed to give you a summary of your ClarityNLP instance at a glance. The Dashboard shows you the following:

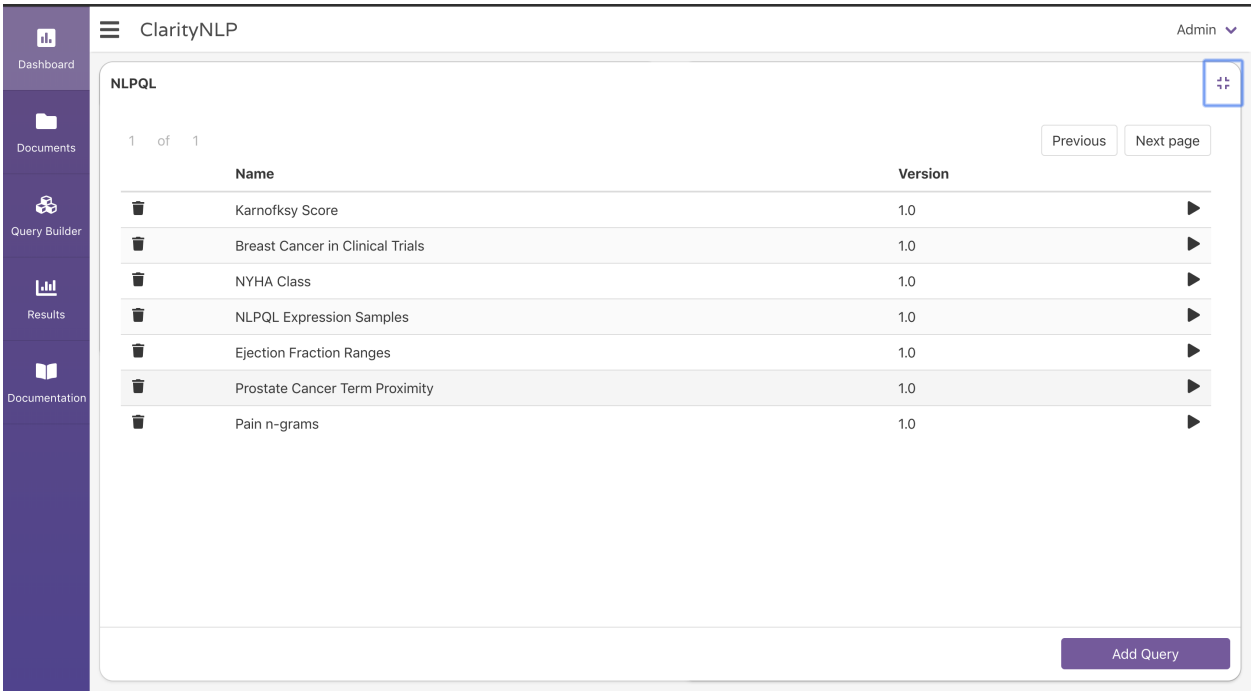
- The document types (and number of each type) stored in your Solr instance
- The NLPQL queries you have created and saved via the Query Builder
- The name, date, runtime, cohort size, and validation status of your NLPQL jobs



To view all of your documents, results, or NLPQL queries, simply click on the expand button at the top right of the corresponding box. This action expands the box to full screen and provides more information on the given field. To return to the default view, click on the collapse button located at the top right corner.


ClarityNLP
Admin
Dashboard
Documents
Query Builder
Results
Documentation
Documents
AACT 4000
PubMed 2000
FDA Drug Labels 1000
MTSAMPLES 15
Add Documents


ClarityNLP
Admin
Dashboard
Documents
Query Builder
Results
Documentation
Results
1 of 1
Previous
Next page
Job
Date
Runtime
n
Accuracy
Ejection Fraction Values
8 Jul 2019 15:07
1:51
212
Validate
Pain n-gram
8 Jul 2019 13:07
IN_PROGRESS
212
Validate
Prostate Cancer Term Proximity
8 Jul 2019 13:07
0:20
30
Validate
Ejection Fraction Values
8 Jul 2019 13:07
1:54
212
Validate
See All Results



Additional Features

If you would like to quickly navigate to the results of one your recent jobs, you can click on that job's row in the results table to go directly to the results in the Results Viewer.

Alternatively, if you would like to open the NLPQL file of a previously saved query, click on the respective row in the NLPQL table to open that query in the Query Builder.

You can run saved queries by pressing the play button (right-pointing arrowhead) in the same row as the query.

You can delete a saved query by pressing the trash can icon next to the name of the query.

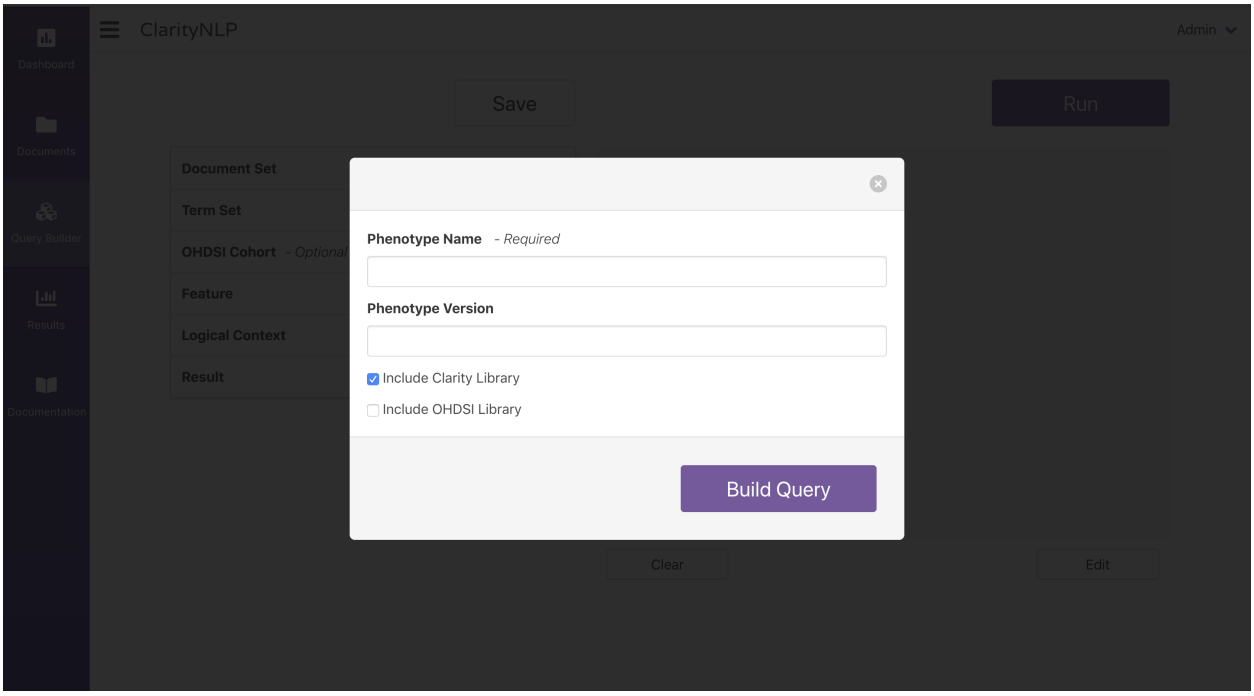
Ingest Client

Query Builder

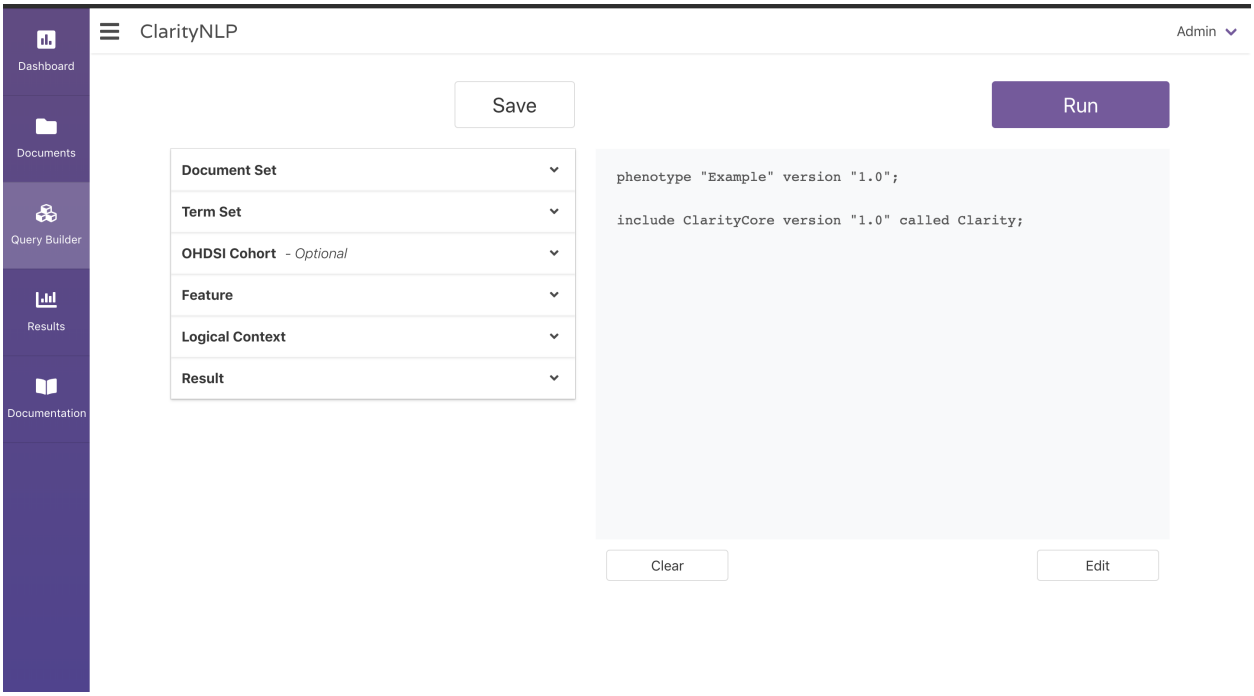
The Query Builder is a tool to assist in the development of NLPQL queries.

When you open the Query Builder, it will prompt you to enter a name and version number for your NLPQL query.

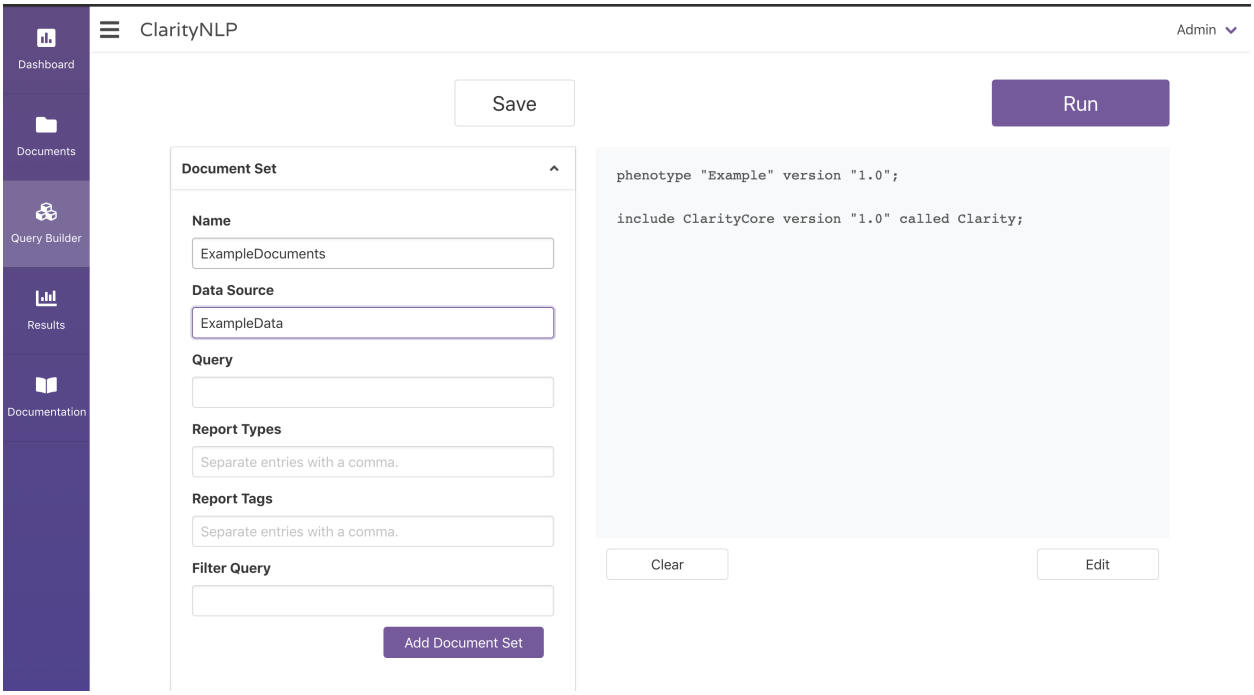
NOTE: a previously-saved query cannot be overwritten. Each query must be saved with a unique version number.

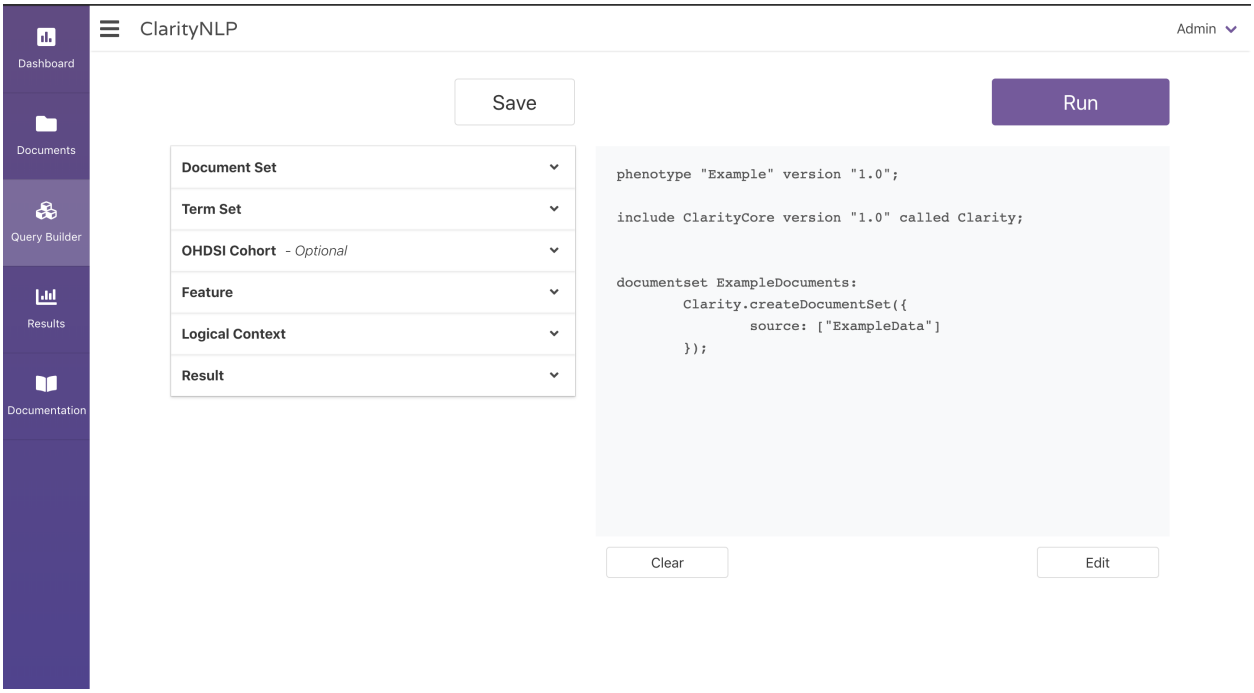


After you have named and versioned your query, you can begin building your query.



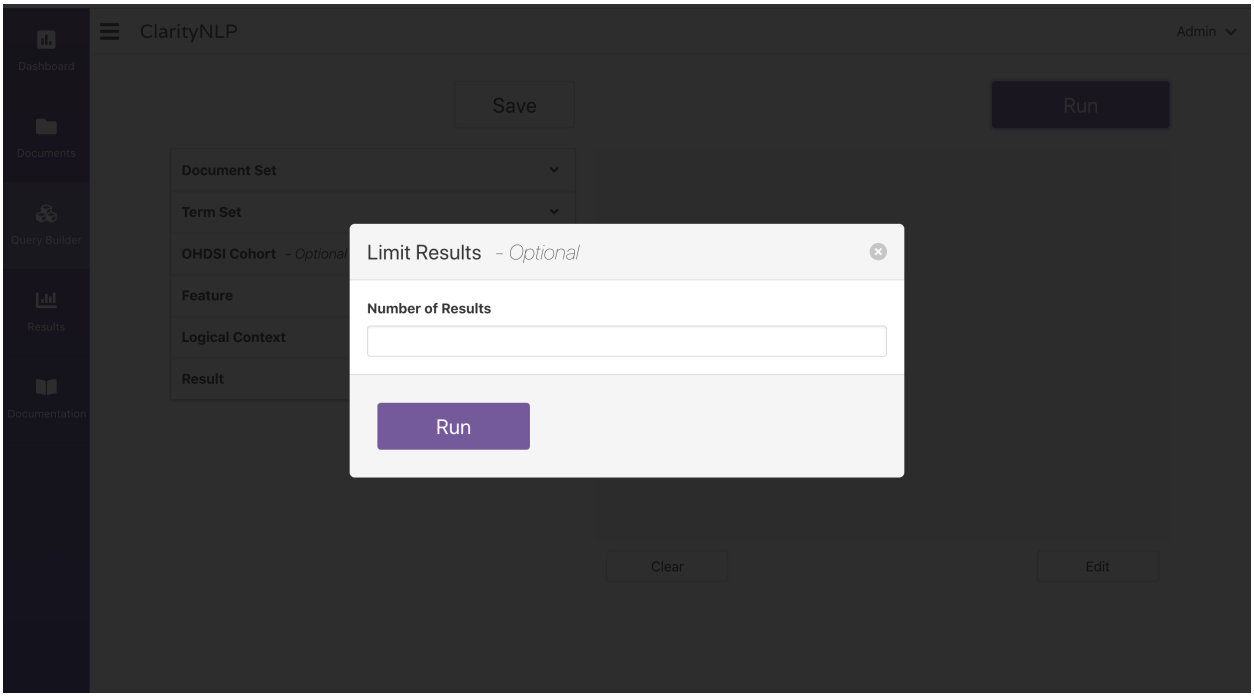
To add a component to the query, simply click on the component you would like to add, fill out the fields that you need, and click the corresponding add button.



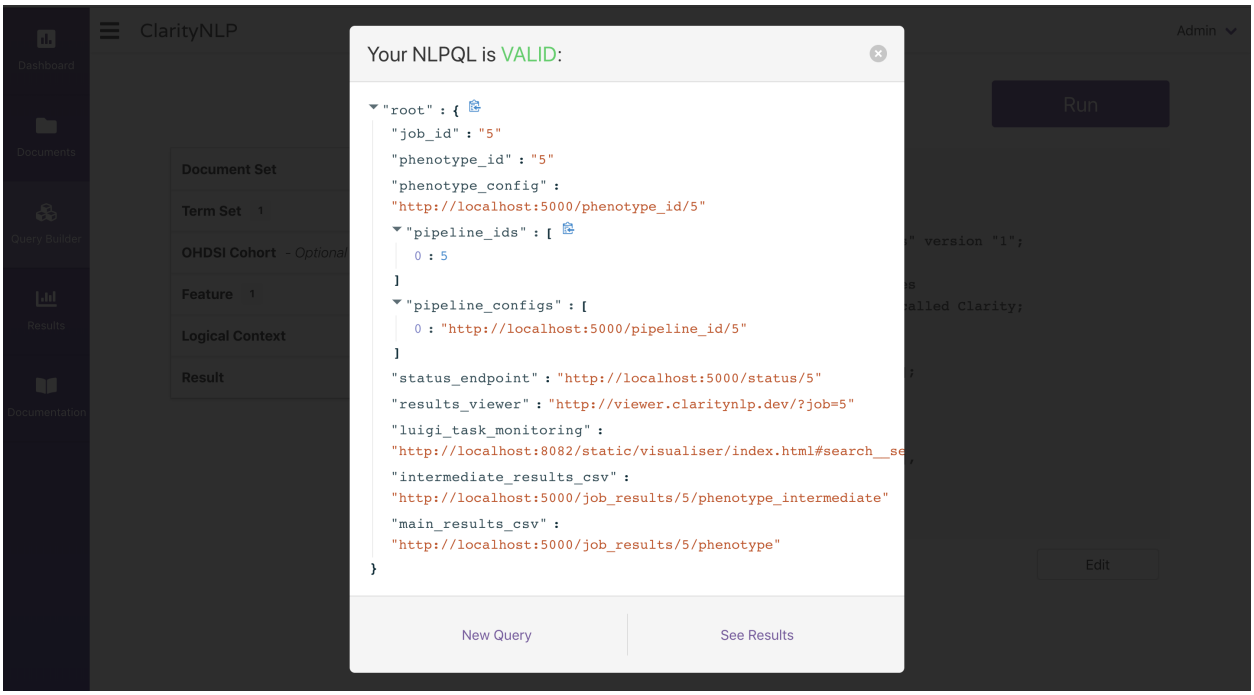


After you finish building your query, you can click the save button just above the dropdowns to save your query indefinitely. You can also run the query by clicking the run button at the top right.

You can specify a limit to the number of documents processed by using an NLPQL `limit` statement. The form in the next image allows you to specify a limit:



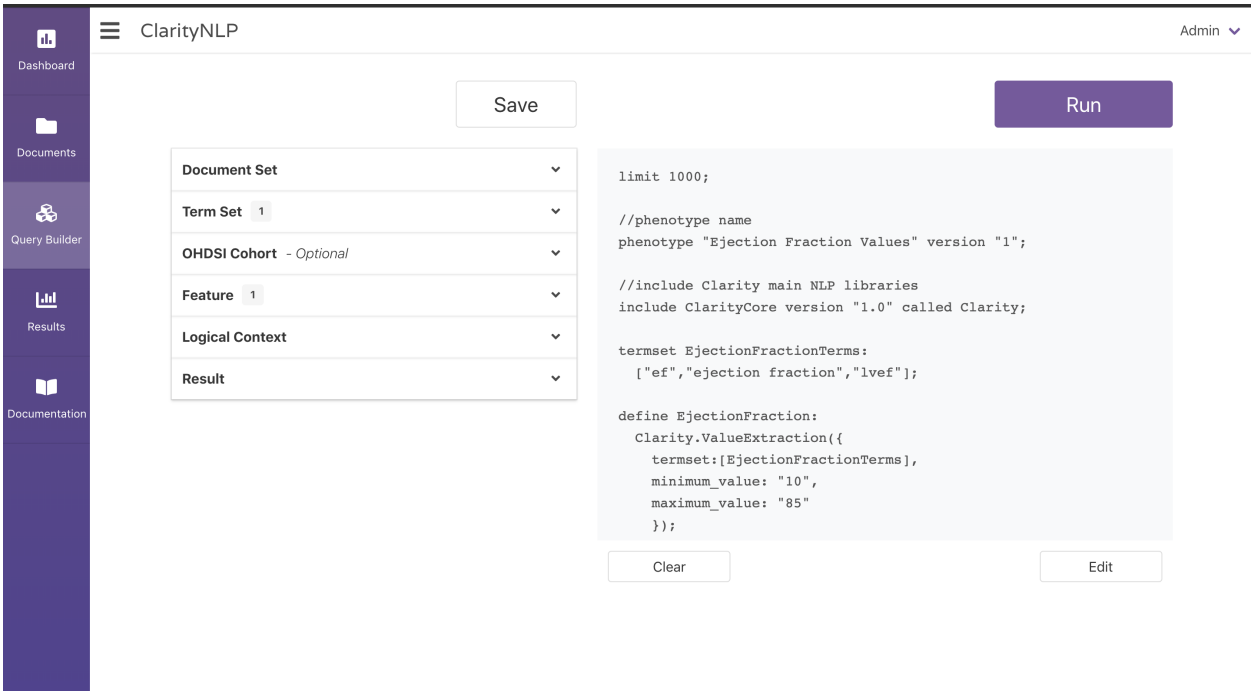
If your NLPQL query passes the validity checks, you will be shown some metadata about your query:



NOTE: If at any point you want to delete your query you can click the clear button at the bottom of the text area to start over.

Loading a Query

If you navigated here from a link or from the Dashboard, your query will automatically load into the text area.



You can edit this query by clicking the edit button below the text area.

To learn more about NLPQL, please see the NLPQL Reference.

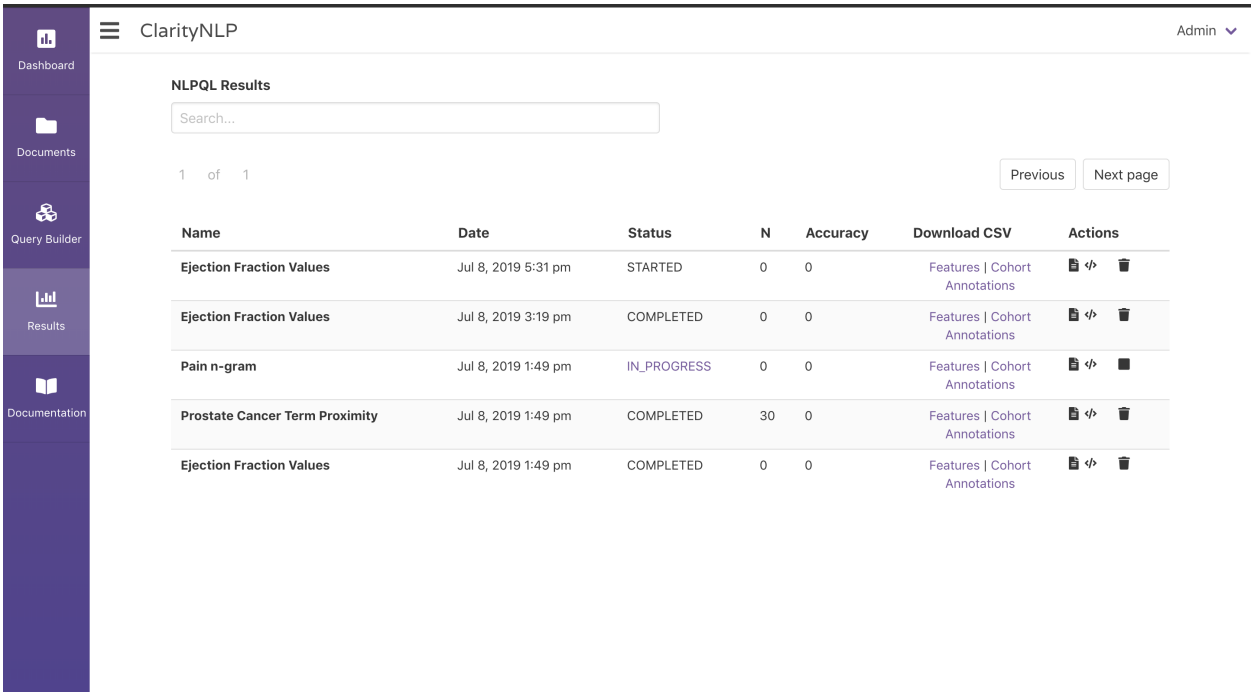
Results Viewer

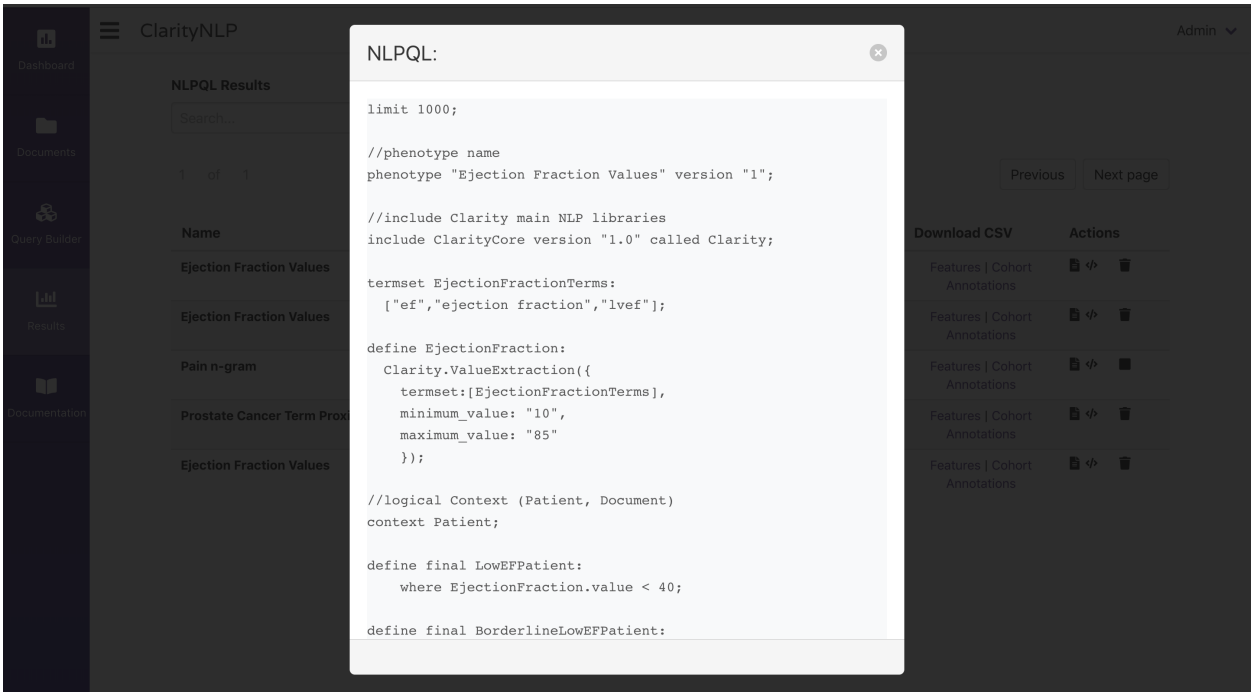
The Results viewer is designed to give you a comprehensive look at the results from a ClarityNLP run.

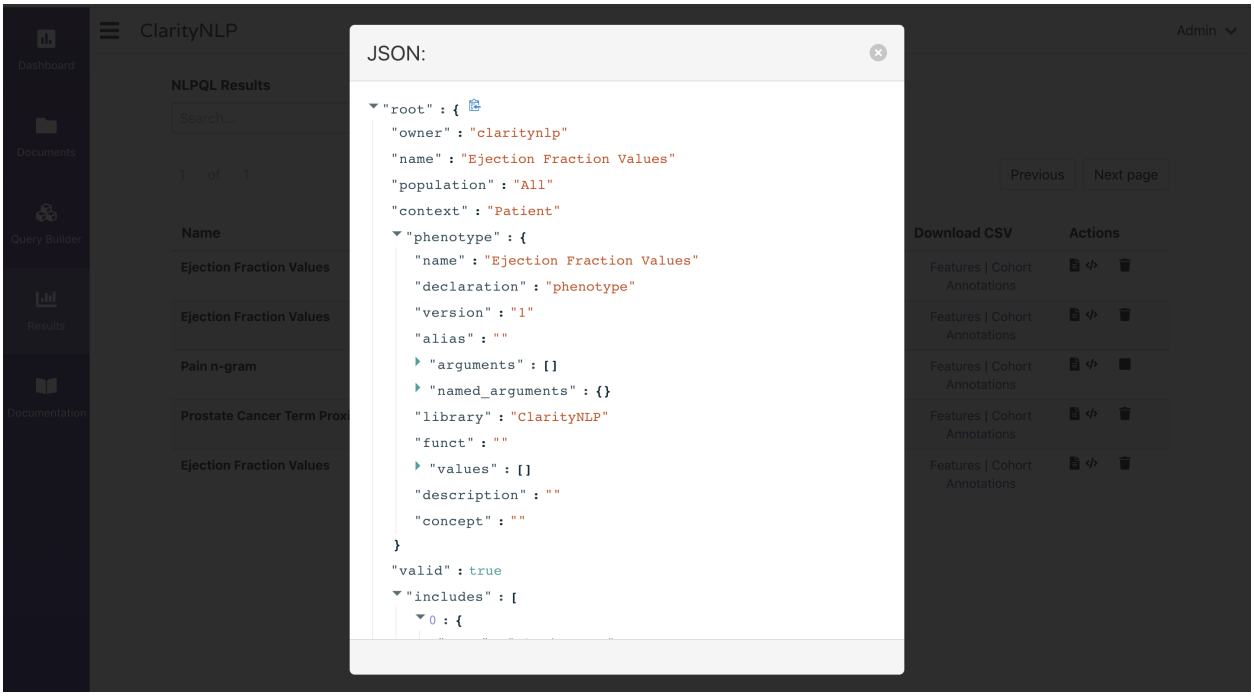
The first screen provides a list of the 20 most recently-submitted jobs. You can navigate the results by using the "Next page" and "Previous" buttons at the top right of the list. Each job has many interactions associated with it, which are:

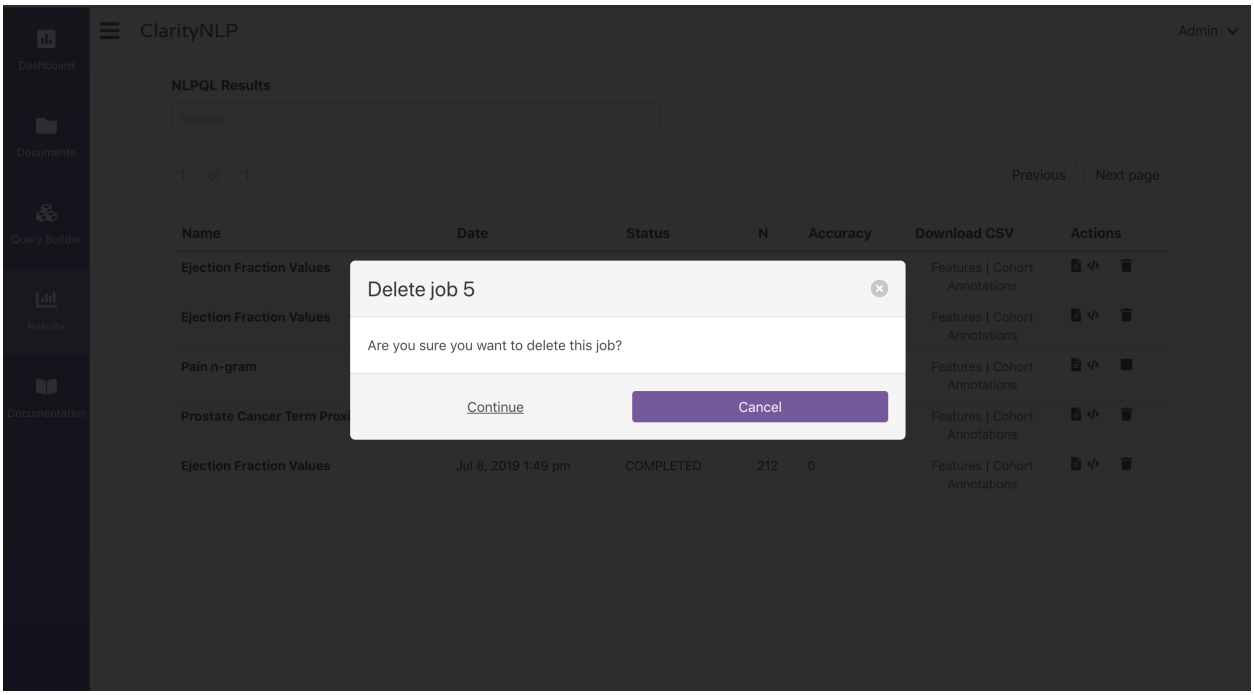
- The name of the query
- The query submission date
- The current status of the job. If the job is not finished, the job status appears as a hyperlink that takes you to the Luigi task viewer for that job.
- The cohort size for that particular query
- The accuracy score from an evaluation of the results
- **Download links for the job that includes CSVs of:**
  - Results
  - Cohort
  - Annotations
- **Actions that can be taken for the job, which include:**
  - Viewing the text representation of the query
  - Viewing the JSON representation of the query
  - Deleting the job

NOTE: Job deletion is permanent and cannot be undone.

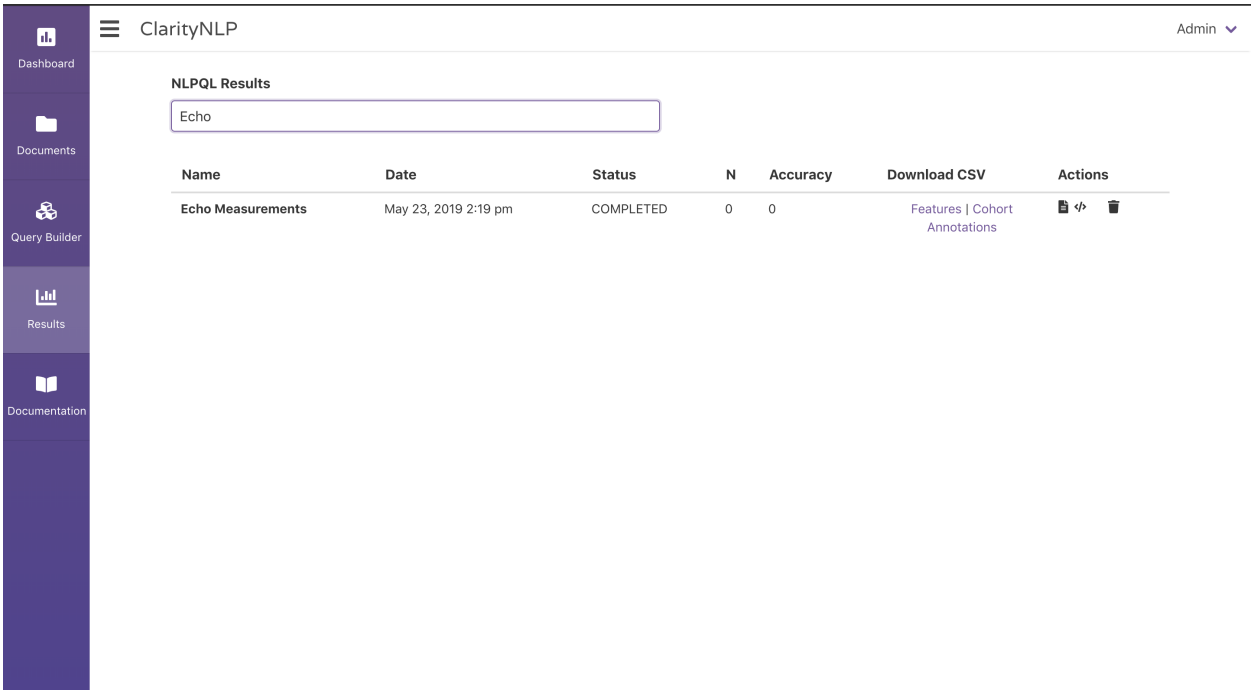




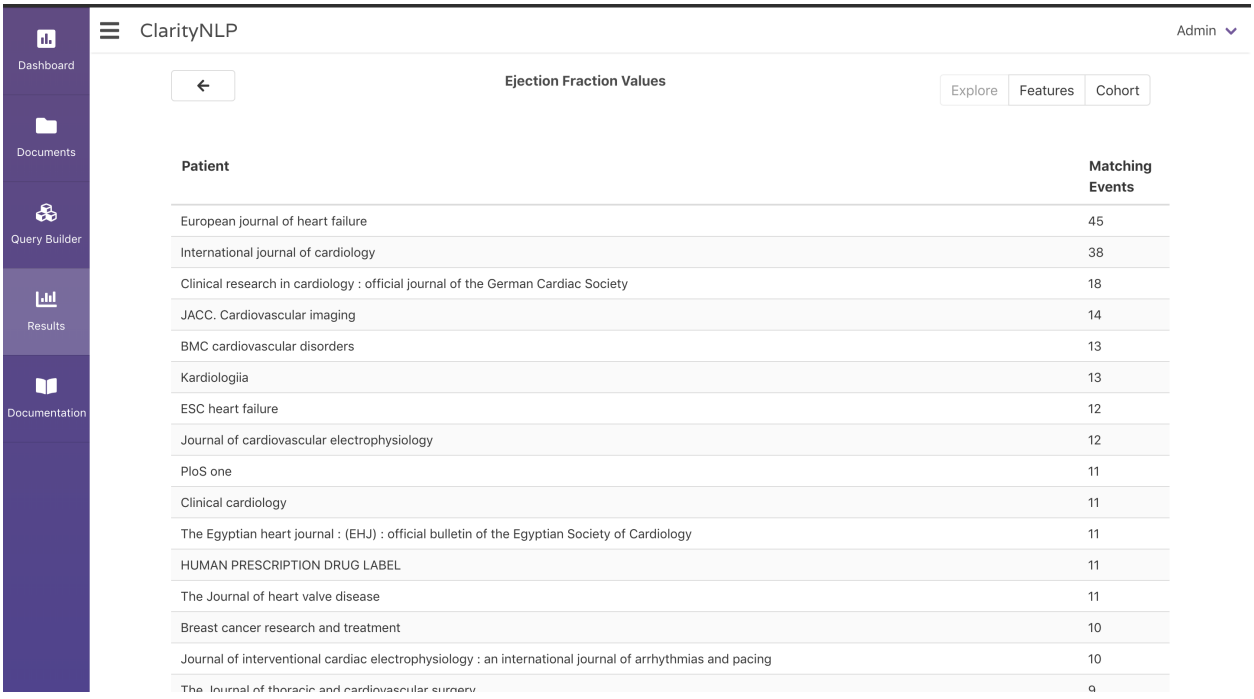




This list is also searchable via terms entered into the text box above the list:

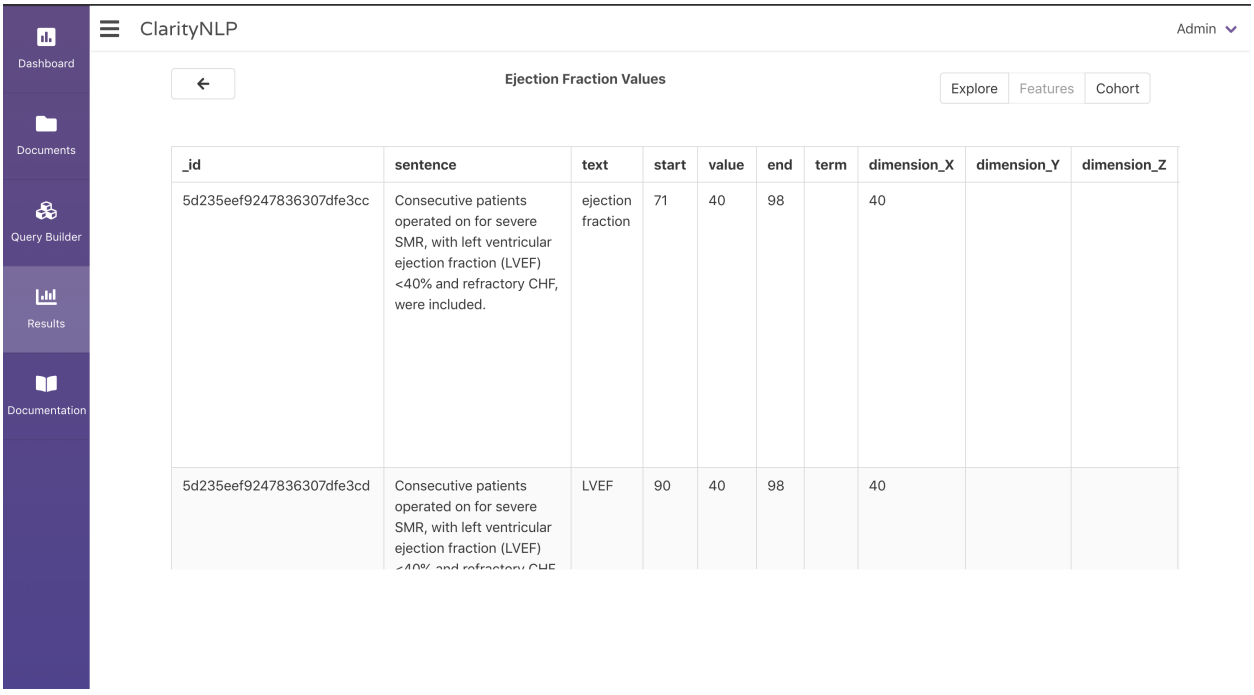


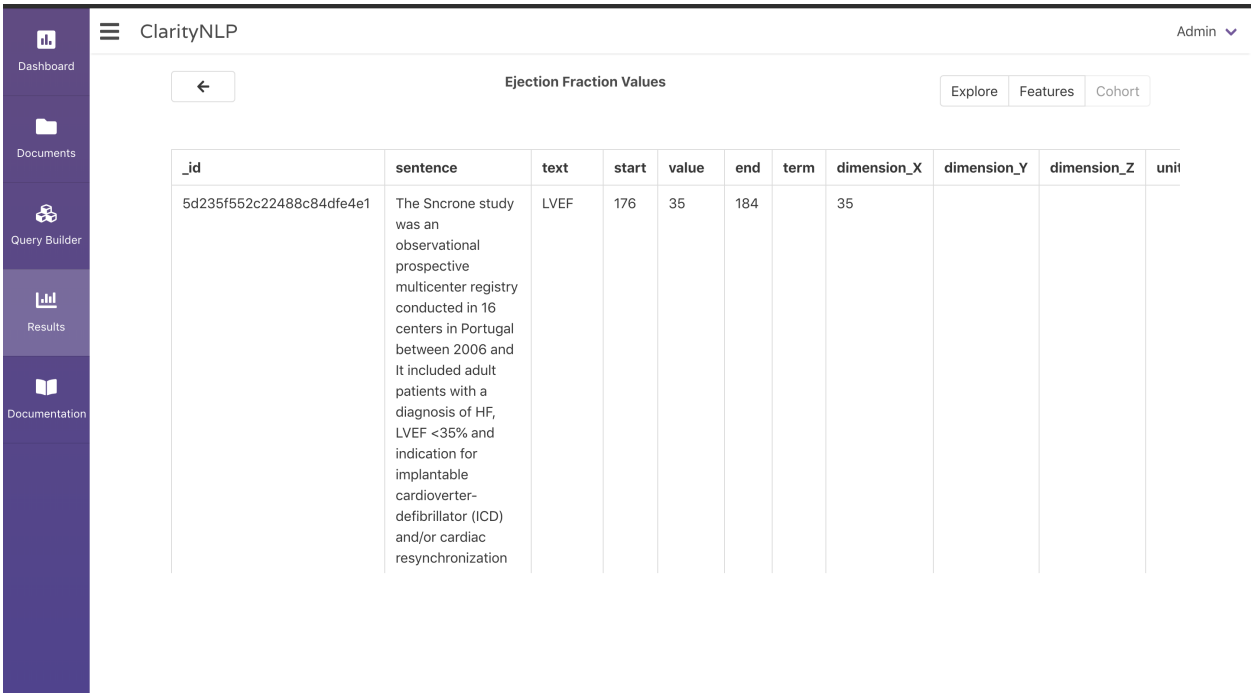
To delve deeper into the results for a job, click on that job's row in the list. This brings you to a screen where you can see individual results from the query. You can also see the number of events that were recognized for each result.



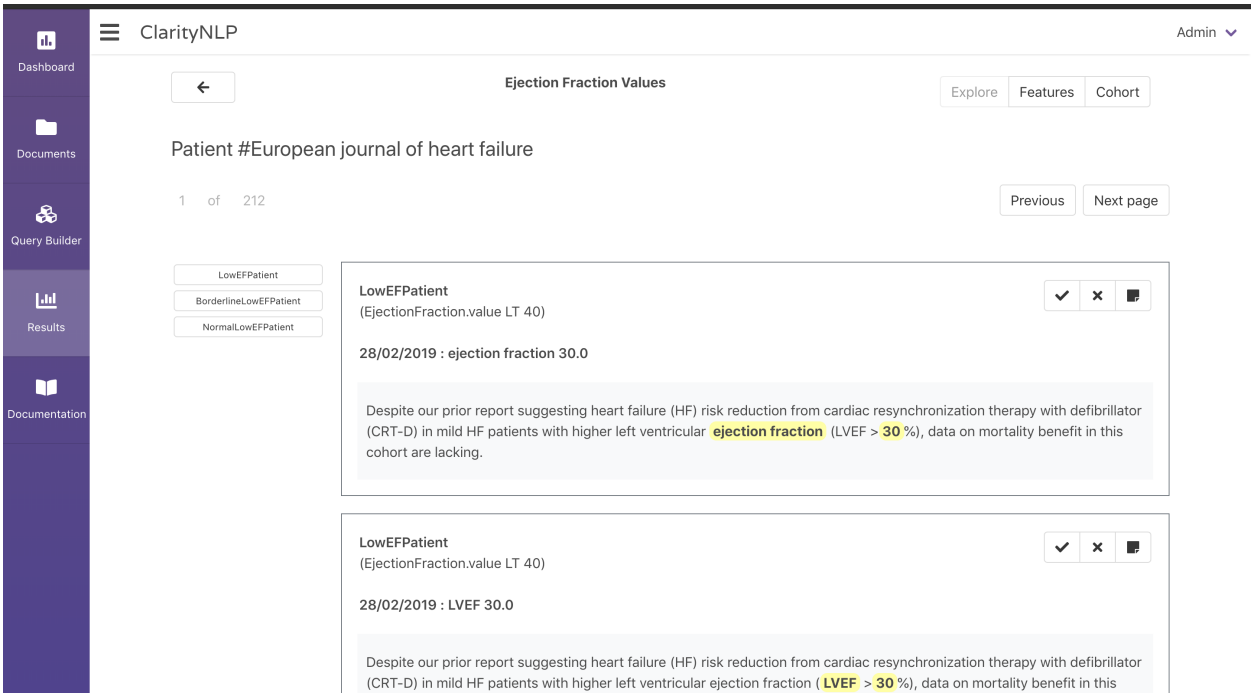
If no results were found for a query, a blank screen will appear.

At the top right of the page, you can cycle through the "Explore", "Feature", and "Cohort" views. The Feature and Cohort views appear as scrollable tables. The Explore view is the default.



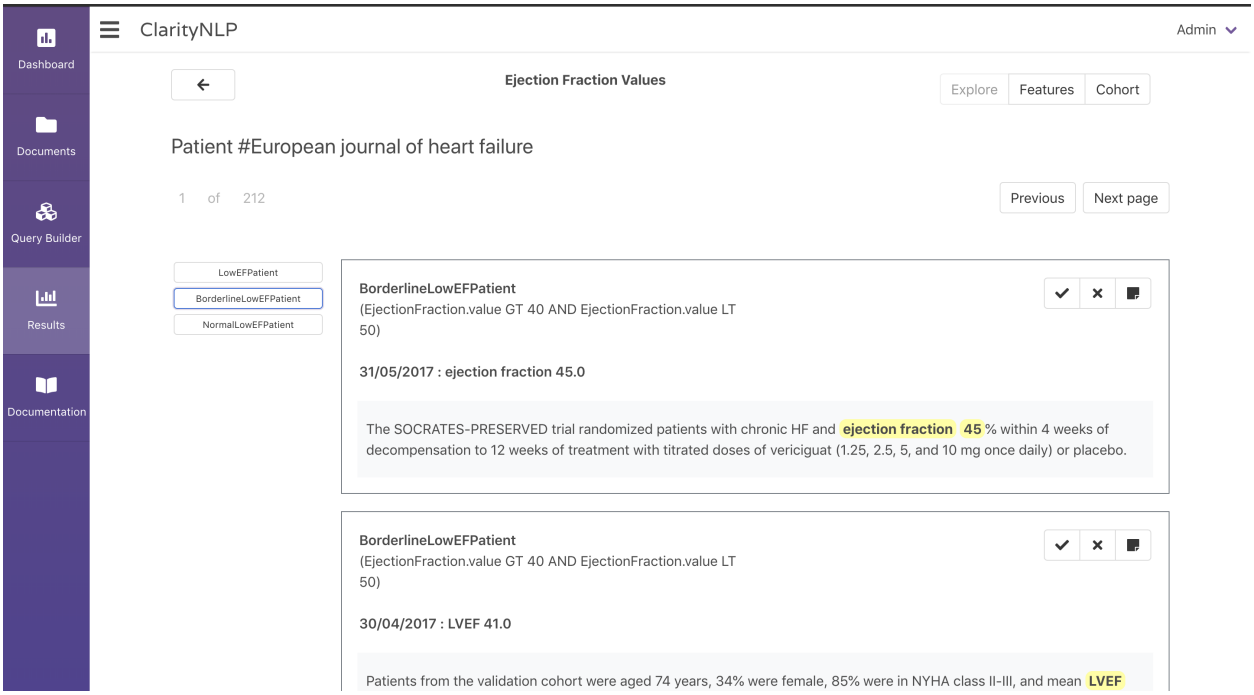


If you want to view the results for a patient, click that patient's row in the list. This will bring you to a screen where you can see highlighted results.



The "Next Page" and "Previous" buttons to the top right will cycle you through the various patients for that job.

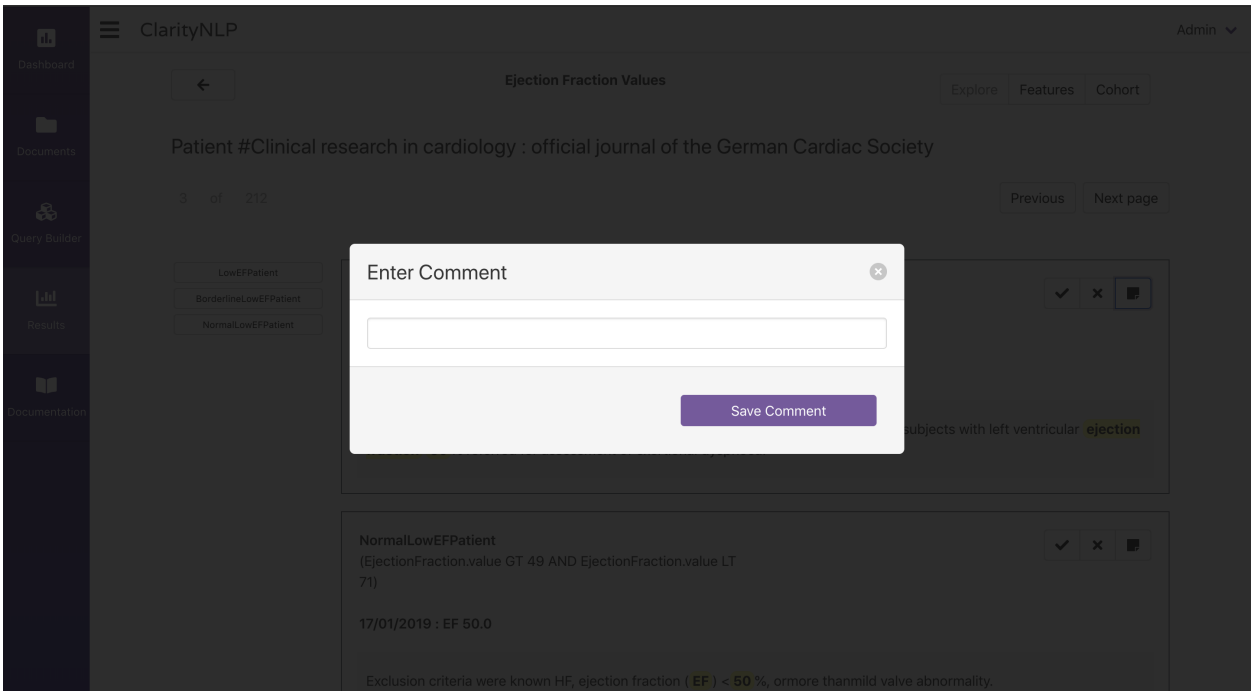
At the left side of the page is a list of the NLPQL features for which results were found. Clicking one of these features will show the results for that feature only. All results are displayed by default.



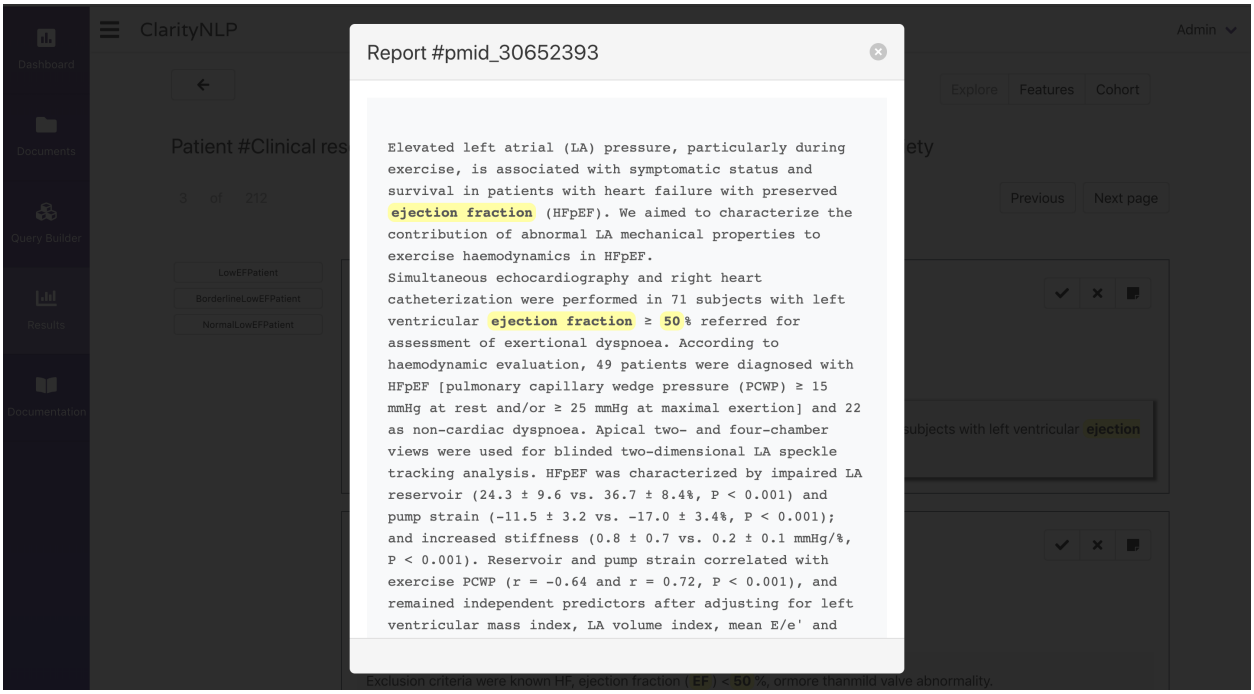
Each result box shows the name of the relevant NLPQL feature, the definition of that feature immediately below it, and an extract from a source document. The extract highlights relevant terms and values associated with the feature.

In the upper right corner of each result box is a set of buttons that can be used to evaluate ClarityNLP's results. You can:

- Click the checkmark if the result is correct
- Click the X if the result is incorrect
- Click the notepad to enter a comment about the result



You can click on the sentence to view the complete source document:



1.2.3 How to Write a Query

ClarityNLP at a Glance

ClarityNLP is designed to simplify the process of analyzing unstructured data (eg. provider notes, radiology reports, pathology results, etc) to find particular data or patients from electronic medical records.

We refer to the definition of what you are trying to find as a *phenotype*. Phenotypes are useful for research, clinical, quality, or payment purposes, because they allow a very explicit definition of the criteria that make up a patient of interest. With ClarityNLP, these criteria can be shared using machine-interpretable code that can be run on any clinical dataset.

How is this accomplished? ClarityNLP uses a query syntax called **Natural Language Processing Query Language (NLPQL)**, based on the CQL syntax from HL7. The ClarityNLP platform provides mapping tools that allow NLPQL phenotypes to run on any dataset.

Let's take a look at how NLPQL works.

Example NLPQL Phenotype Walkthrough

Imagine you have a dataset with thousands of clinical documents and would like to extract a particular data element. You can create an NLPQL file to specify what you would like to extract.

Here is a basic NLPQL phenotype that extracts Temperature values from Nursing notes.

```
phenotype "Patient Temperatures" version "2";

include ClarityCore version "1.0" called Clarity;

documentset NursingNotes:
```

(continues on next page)

(continued from previous page)

```
   Clarity.createReportTagList(["Nurse"]);

termset TemperatureTerms:
   ["temp","temperature","t"];

 define Temperature:
   Clarity.ValueExtraction({
     termset:[TemperatureTerms],
     documentset: [NursingNotes],
     minimum_value: "96",
     maximum_value: "106"
     });

 define final hasFever:
     where Temperature.value >= 100.4;
```

Let's break down the NLPQL above.

Phenotype Name

```
phenotype "Patient Temperatures" version "2";
```

Every ClarityNLP phenotype definition needs a name. We give it a name (and optionally a version) using the `phenotype` command. Here, we are just declaring that our phenotype will be called "Patient Temperatures".

Libraries

```
include ClarityCore version "1.0" called Clarity;
```

NLPQL is designed to be extensible and make it easy for developers to build new NLP algorithms and run them using the ClarityNLP platform. A common paradigm for making software extensible is the use of libraries. Using the `include` command, we are saying to include the core Clarity library which has lots of handy commands and NLP algorithms built-in. The `called` phrase allows us to select a short name to refer to the library in the NLPQL that follows. In this case, we have selected to call it "Clarity".

Document Sets

```
documentset NursingNotes:
   Clarity.createReportTagList(["Nurse"]);
```

Document sets are lists of document types that you would like ClarityNLP to process. (If no document sets are created, ClarityNLP will simply analyze all documents in your repository.) Built into the Clarity core library is the `createReportTagList` function, which allows you to enumerate a set of document type tags from the LOINC document ontology. Typically, these tags are assigned to your documents at the time of ingestion through use of the Report Type Mapper.

In this case, we have declared a document set called "Nursing Notes" and included in it all documents with the Nurse tag. We could have selected another provider type (eg. Physician), a specialty type (eg. Endocrinology), a setting type (eg. Emergency Department), or a combination such as `["Physician","Emergency Department"]`.

```
documentset AmoxDischargeNotes:
     Clarity.createDocumentSet({
          "report_types":["Discharge summary"],
          "report_tags": [],
          "filter_query": "",
          "query":"report_text:amoxicillin"});
```

ClarityNLP provides an additional document set, `createDocumentSet`, which provides more control over document section, allowing users to select report tags or report types, and provides flexibility to write custom queries.

Term Sets

```
termset TemperatureTerms:
   ["temp","temperature","t"];
```

Term sets are lists of terms or tokens you would like to input into an NLP method. You can create these lists manually (as shown in this example) or generate them based on ontologies. Furthermore you can extend termsets with synonyms and lexical variants.

In this case, we have created a term set called "TemperatureTerms" and included three common ways temperature is referenced in a clinical note ("temperature", "temp", and "t").

Phenotype Features

Features are the clinical elements that you wish to find and analyze in order to identify your patients of interest. Features specify an *NLPQL task* you'd like to run as well as optional parameters such as document sets, term sets, patient cohorts, and more. See the NLPQL examples to get a better sense of how different features can be created.

We have two features in our example NLPQL. Let's take a look at each.

```
define Temperature:
   Clarity.ValueExtraction({
     termset:[TemperatureTerms],
     documentset: [NursingNotes],
     minimum_value: "96",
     maximum_value: "106"
     });
```

Features are specified in NLPQL using the `define` keyword followed by a feature name and a function. In this case, we are assigning the name "Temperature" to the output of a particular NLP method that is included in the Clarity core library called *Value Extraction*. (This could just as easily have been an NLP method from another Python library or an external API using *External NLP Method Integration*.)

In the example, we provide the Value Extraction method with a set of parameters including our document set ("NursingNotes"), term set ("TemperatureTerms"), and min/max values to include in the temperature results. The accuray of this definition for temperature can be evaluated using the ClarityNLP validation framework, which is a feature built into the *Results Viewer*.

Now on to the second feature in the example:

Final Features

```
define final hasFever:
    where Temperature.value >= 100.4;
```

With this statement, we are creating a new feature called "hasFever" that includes any patients with a temperature value greater than 100.4. There are two things to note about this syntax.

- `final` A phenotype may involve the creation of numerous intermediate features that are extracted by NLP processes but are not themselves the final result of the analysis. For example, we may be interested only in patients with a fever, rather than any patient who has a temperature value recorded. The *final* keyword allows us to indicate the final output or outputs of the phenotype definition.
- `value` Every NLP method returns a result. The specific format and content of these results will vary by method. As a convenience, ClarityNLP returns a `value` parameter for most methods. The *Value Extraction* method used here also returns several other parameters. ClarityNLP is flexible in that it can take any parameter you provide and perform operations on it. However, this will only work if the method being called returns that parameter. Please consult the documentation for individual methods to see what parameters can be referenced.

Running NLPQL Queries

In the full guide, we will walk you through the steps of ingesting and mapping your own data. Once in place, you will be able to run queries by hitting the *nlpql API endpoint* on your local server or by visiting `<your_server url>:5000/nlpql`. But to run a quick test, feel free to use our NLPQL test page.

Next Steps

The next steps for you are to *install ClarityNLP*, follow through some of our Cooking with Clarity tutorials to learn how to create a full-blown ClarityNLP project, and join our channel on Slack. Thanks for your interest!

1.2.4 Basic NLP Phenotype Examples

Overview

Before going through these examples, make sure to review the NLPQL walkthrough to get an understanding of the general components of NLPQL. For this set of examples, we will be focusing on extracting data relevant to congestive heart failure.

Note we recommend prepending each query with `limit 100;` which keeps the job small and allows you to test queries without taking up a lot of time and compute resources. Once you have developed a query and want to scale it to the full dataset, simply remove the `limit` statement.

All of the sample results shown here are from the de-identified MIMIC III dataset.

Example 1: Finding Symptoms of a Disease

In this first example, we are looking for certain symptoms of congestive heart failure likely to be found only in the clinical notes. Specifically we are looking for orthopnea and paroxysmal nocturnal dyspnea (PND).

Using TermFinder

```
limit 100;

//phenotype name
phenotype "Orthopnea" version "2";

//include Clarity main NLP libraries
include ClarityCore version "1.0" called Clarity;
```

(continues on next page)

(continued from previous page)

```
termset Orthopnea:
  ["orthopnea","orthopnoea","PND"];

define hasOrthopnea:
  Clarity.TermFinder({
    termset:[Orthopnea]
    });
```

Here we have simply defined a set of terms we are interested and lumped them into a `termset` called *Orthopnea*. We could have named this termset anything.

The `TermFinder` function simply takes in that list of terms and finds all documents with these terms, without any additional filtering. Here are example results.

experiencer	negation	sentence	section
Patient	Negated	Cardiac review of systems is notable for absence of paroxysmal nocturnal dyspnea, ort	HISTORY_PRESENT_ILLNESS
Patient	Negated	No orthopnea or paroxysmal nocturnal dyspnea.	HISTORY_PRESENT_ILLNESS
Patient	Affirmed	Per patient, + for orthopnea and increasing SOB	HISTORY_PRESENT_ILLNESS
Patient	Negated	The patient denied any symptoms of shortness of breath, orthopnea, claudication, or	HISTORY_PRESENT_ILLNESS

Orthopnea Sample Results

As you can see, while the TermFinder was helpful in finding mentions of our Orthopnea terms, much of what was found were actually negative mentions (ie, the patient did *not* have the symptom). So ClarityNLP lets you set a variety of constraints around TermFinder, for example limiting results to particular sections of the note or just to affirmed mentions.

```
define hasOrthopnea:
  Clarity.TermFinder({
    termset:[Orthopnea],
    negated:"Affirmed",
    sections:["CHIEF_COMPLAINT","HISTORY_PRESENT_ILLNESS"]
    });
```

But because in most situations we need to find positive mentions that are current and relevant to the patient, ClarityNLP has a convenient function called `ProviderAssertion` that allows you to bypass entering all the typical parameters. Here is a simple example.

Using ProviderAssertion

```
limit 100;
//phenotype name
phenotype "Orthopnea" version "2";

//include Clarity main NLP libraries
include ClarityCore version "1.0" called Clarity;

termset Orthopnea:
  ["orthopnea","orthopnoea","PND"];

define hasOrthopnea:
  Clarity.ProviderAssertion({
    termset:[Orthopnea]
    });
```

As you can see, the results are now limited to just positive mentions.

experiencer	negation	section	sentence
Patient	Affirmed	HISTORY_PRESENT_ILLNESS	55 y/o man with diabetes and heel ulcer with orthopnea.
Patient	Affirmed	HISTORY_PRESENT_ILLNESS	**]-year-old woman with a history of coronary artery disease, paroxysmal atrial fibrillation, orthopnea, severe PND.I
Patient	Affirmed	HISTORY_PRESENT_ILLNESS	63-year-old woman with shortness of breath, PND, orthopnea, question edema or pneumonia.
Patient	Affirmed	HISTORY_PRESENT_ILLNESS	80-year-old female with orthopnea and shortness of breath.
Patient	Affirmed	HISTORY_PRESENT_ILLNESS	Metastatic hepatoma, now with ____ orthopnea.
Patient	Affirmed	HISTORY_PRESENT_ILLNESS	67-year-old female with worsening shortness of breath and orthopnea.

Example 1.1 Results

Example 2: Extracting Quantitative Values

In this example, we will be searching for ejection fraction values using a very simple algorithm. Specifically, we will be looking for certain terms and subsequent values that would be typical for EF values. There are many more sophisticated methods to find ejection fraction (e.g Kim et al). Our goal in this example is to provide you familiarity with the use of the ClarityNLP `ValueExtraction` functionality.

```
limit 100;
//phenotype name
phenotype "Ejection Fraction Values" version "1";

//include Clarity main NLP libraries
include ClarityCore version "1.0" called Clarity;

termset EjectionFractionTerms:
  ["ef","ejection fraction","lvef"];

define EjectionFraction:
  Clarity.ValueExtraction({
    termset:[EjectionFractionTerms],
    minimum_value: "10",
    maximum_value: "85"
    });
```

ValueExtractor	left ventricular ejection fraction is 66%.	ejection fraction	66
ValueExtractor	lvef 66%.	lvef	66
ValueExtractor	lv with ef of approximately 30%.	ef	30
ValueExtractor	new cardiomyopathy ef 35%.	ef	35
ValueExtractor	overall left ventricular systolic function is mildly depressed(lvef= 45 %)	lvef	45
ValueExtractor	left ventricular ejection fraction is 54%.	ejection fraction	54
ValueExtractor	right ventricular ejection fraction is 53%	ejection fraction	53

Example 2.1 Results

If you wanted to find only low ejection fractions, you could do this in two ways. The first is by modifying the min and max parameters. For example:

```
define EjectionFraction:
  Clarity.ValueExtraction({
    termset:[EjectionFractionTerms],
    maximum_value: "30"
    });
```

This will filter your results to only those <30%.

Example 3: Extracting Non-Quantitative Values

In some cases you may want to extract data points that are values but not numeric. A good example is CHF class. Below is an example of NLPQL to pull out NYHA classifications.

```
limit 100;
//phenotype name
phenotype "NYHA Class" version "1";

//include Clarity  main NLP libraries
include ClarityCore version "1.0" called Clarity;

termset NYHATerms:
  ["nyha"];

define NYHAClass:
  Clarity.ValueExtraction({
    termset:[NYHATerms],
    enum_list: ["3","4","iii","iv"];
    });
```

Looking up more stuff

Note: we recommend prepending each query with `limit 100;` which keeps the job small and allows you to test queries without taking up a lot of time and compute resources. Once you have developed a query and want to scale it to the full dataset, simply remove the `limit` statement.

1.3 Developer Guide

1.3.1 For Algorithm Developers

Technical Overview

Technologies We Use

ClarityNLP is built on several popular open-source projects. In this section we provide a brief overview of each project and describe how it is used by ClarityNLP.

Docker

Docker uses *operating-system-level virtualization* to provide a means of isolating applications from each other and controlling their access to system resources. Isolated applications run in restricted environments called *containers*. A container includes the application and all dependencies so that it can be deployed as a self-contained unit.

ClarityNLP can be deployed as a set of Docker containers. The secure OAuth2-based server configuration assumes this deployment mechanism. You can find out more about the ClarityNLP setup options and our use of Docker in our *setup documentation*.

Solr

Apache Solr is an enterprise search platform with many advanced features including fault tolerance, distributed indexing, and the ability to scale to billions of documents. It is fast, highly configurable, and supports a wide range of user customizations.

ClarityNLP uses Solr as its primary document store. Any documents that ClarityNLP processes must be retrieved from Solr. We provide instructions on how to ingest documents into Solr. We also provide some python scripts to help you with common data sets. See our document ingestion documentation for more.

PostgresSQL

PostgreSQL is one of the leading open-source relational database systems, distinguished by its robust feature set, ACID compliance, and excellent performance. ClarityNLP uses Postgres to store data required to manage each NLPQL job. Postgres is also used to store a large amount of medical vocabulary and concept data.

MongoDB

MongoDB is a popular NoSQL document store. A mongo *document* is a JSON object with user-defined fields and values. There is no rigid structure imposed on documents. Multiple documents form groups called *collections*, and one or more collections comprise a *database*.

ClarityNLP uses Mongo to store the results that it finds. The ClarityNLP built-in and custom tasks all define result documents with fields meaningful to each task. ClarityNLP augments the result documents with additional job-specific fields and stores everything in a single collection.

ClarityNLP also evaluates *NLPQL expressions* by translating them into a MongoDB aggregation pipeline.

NLP Libraries (spaCy, textacy, nltk)

The natural language processing libraries spaCy and nltk provide implementations of the fundamental NLP algorithms that ClarityNLP needs. These algorithms include sentence segmentation, part-of-speech tagging, and dependency parsing, among others. ClarityNLP builds its NLP algorithms on top of the foundation provided by spaCy and nltk.

Textacy is a higher-level NLP library built on spaCy. ClarityNLP uses textacy for its *Clarity.ngram* task and for computing text statistics with *Clarity.TextStats*.

Luigi

Luigi is a python library that manages and schedules pipelines of batch processes. A *pipeline* is an ordered sequence of tasks needed to compute a result. The tasks in the pipeline can have *dependencies*, which are child tasks that must run and finish before the parents can be scheduled to run. Luigi handles the task scheduling, dependency management, restart-on-failure, and other necessary aspects of managing these pipelines.

The *NLPQL Reference* defines a set of core and custom tasks that comprise the data processing capabilities of ClarityNLP. ClarityNLP uses Luigi to schedule and manage the execution of these tasks.

Flask

Flask is a "micro" framework for building Web applications. Flask provides a web server and a minimal set of core features, as well as an extension mechanism for including features found in more comprehensive Web frameworks.

The ClarityNLP component that provides the *NLP Web APIs* is built with Flask.

Redis

Redis is an in-memory key-value store that is typically used as a fast cache for frequently-accessed data. The values mapped to each key can either be strings or more complex data structures. Redis supports many advanced features such as partitioning and time-based key expiration.

ClarityNLP uses Redis as a fast query cache.

Pandas

Pandas is a python library for data analysis, with particular strengths in manipulating tabular and labeled data. It provides data structures and methods for doing operations that one would typically use a spreadsheet for. It provides a powerful I/O library and integrates fully with the python machine learning, data analysis, and visualization stack.

ClarityNLP uses pandas for some I/O operations and for various forms of data manipulation.

Client-side Libraries (React, Sails)

TBD

Solr Setup and Configuration

Data types

We use standard Solr data types with one custom data type, `searchText`. `searchText` is a text field, tokenized on spaces, with filtering to support case insensitivity.

Fields

All documents in ClarityNLP are stored in Solr. These are the minimal required fields:

```
{
        "report_type":"Report Type",
        "id":"1",
        "report_id":"1",
        "source":"My Institution",
        "report_date":"1970-01-01T00:00:00Z",
        "subject":"the_patient_id_or_other_identifier",
        "report_text":"Report text here"
    }
```

`id` and `report_id` should be unique in the data set, but can be equal. `report_text` should be plain text. `subject` is generally the patient identifier, but could also be some other identifier, such as `drug_name`. `source` is generally your institution or the name of the document set.

Additional fields can be added to store additional metadata. The following fields are allowable as dynamic fields:

- `*_section` (searchText); e.g. `past_medical_history_section` (for indexing specific sections of notes)
- `*_id` (long) e.g.`doctor_id` (any other id you wish to store)
- `*_ids` (long, multiValued) e.g. `medication_ids` (any other id as an array)

- `*_system` (string) e.g. `code_system` (noting any system values)
- `*_attr` (string) e.g.`clinic_name_attr` (any single value custom attribute)
- `*_attrs` (string, multiValued) e.g. `insurer_names` (any multi valued custom attribute)

Custom Solr Setup

This should be completed for you if you are using Docker. However, here are the commands to setup Solr.

- Install Solr
- Setup custom tokenized field type:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
      "add-field-type" : {
         "name":"searchText",
         "class":"solr.TextField",
         "positionIncrementGap":"100",
         "analyzer" : {
            "charFilters":[{
               "class":"solr.PatternReplaceCharFilterFactory",
               "replacement":"$1$1",
               "pattern":"([a-zA-Z])\\\\1+" }],
            "tokenizer":{
               "class":"solr.WhitespaceTokenizerFactory" },
            "filters":[{
               "class":"solr.WordDelimiterFilterFactory",
               "preserveOriginal":"0" },
               {"class": "solr.LowerCaseFilterFactory"
               }]}}
    }' http://localhost:8983/solr/report_core/schema
```

- Add standard fields (Solr 6):

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{"name":"report_date","type":"date","indexed":true,"stored":true,
↪"default":"NOW"},
  "add-field":{"name":"report_id","type":"string","indexed":true,"stored":true},
  "add-field":{"name":"report_text","type":"searchText","indexed":true,"stored":true,
↪"termPositions":true,"termVectors":true,"docValues":false,"required":true},
  "add-field":{"name":"source","type":"string","indexed":true,"stored":true},
  "add-field":{"name":"subject","type":"string","indexed":true,"stored":true},
  "add-field":{"name":"report_type","type":"string","indexed":true,"stored":true}
}' http://localhost:8983/solr/report_core/schema
```

- Add standard fields (Solr 7 and later):

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{"name":"report_date","type":"pdate","indexed":true,"stored":true,
↪"default":"NOW"},
  "add-field":{"name":"report_id","type":"string","indexed":true,"stored":true},
  "add-field":{"name":"report_text","type":"searchText","indexed":true,"stored":true,
↪"termPositions":true,"termVectors":true,"docValues":false,"required":true},
  "add-field":{"name":"source","type":"string","indexed":true,"stored":true},
  "add-field":{"name":"subject","type":"string","indexed":true,"stored":true},
  "add-field":{"name":"report_type","type":"string","indexed":true,"stored":true}
}' http://localhost:8983/solr/report_core/schema
```

- Add dynamic fields (Solr 6):

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-dynamic-field":{"name":"*_section","type":"searchText","indexed":true,"stored
↪":false},
  "add-dynamic-field":{"name":"*_id","type":"long","indexed":true,"stored":true},
  "add-dynamic-field":{"name":"*_ids","type":"long","multiValued":true,"indexed":true,
↪"stored":true},
  "add-dynamic-field":{"name":"*_system","type":"string","indexed":true,"stored":true}
↪,
  "add-dynamic-field":{"name":"*_attr","type":"string","indexed":true,"stored":true},
  "add-dynamic-field":{"name":"*_attrs","type":"string","multiValued":true,"indexed
↪":true,"stored":true}
}' http://localhost:8983/solr/report_core/schema
```

- Add dynamic fields (Solr 7 and later):

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-dynamic-field":{"name":"*_section","type":"searchText","indexed":true,"stored
↪":false},
  "add-dynamic-field":{"name":"*_id","type":"plong","indexed":true,"stored":true},
  "add-dynamic-field":{"name":"*_ids","type":"plongs","multiValued":true,"indexed
↪":true,"stored":true},
  "add-dynamic-field":{"name":"*_system","type":"string","indexed":true,"stored":true}
↪,
  "add-dynamic-field":{"name":"*_attr","type":"string","indexed":true,"stored":true},
  "add-dynamic-field":{"name":"*_attrs","type":"strings","multiValued":true,"indexed
↪":true,"stored":true}
}' http://localhost:8983/solr/report_core/schema
```

- Ingest data

Deleting documents

These commands will permanently delete your documents; use with caution.

Delete documents based on a custom query:

```
curl "http://localhost:8983/solr/report_core/update?commit=true" -H "Content-Type:␣
↪text/xml" --data-binary '<delete><query>source:"My Source"</query></delete>'
```

Delete all documents:

```
curl "http://localhost:8983/solr/report_core/update?commit=true" -H "Content-Type:␣
↪text/xml" --data-binary '<delete><query>*:*</query></delete>'
```

Pipelines

Pipelines are the lowest level type jobs that can be run with Luigi and ClarityNLP. Generally they have one purpose such as finding provider assertions or extracting temperature measurements. NLPQL is generally composed of one or more pipelines, so usually pipelines don't need to be run standalone, but can be for testing purposes. They can be run from the command line through Luigi (see below), or via POSTing pipeline JSON to the endpoint `http://nlp-api:5000/pipeline`.

Running a standalone pipeline from the command line

```
PYTHONPATH='.' luigi --module luigi_pipeline NERPipeline --pipeline 1 --job 1234 --
↪owner user
```

Utility Algorithms

Section Tagging

Overview

The section tagger ingests clinical documents and uses textual clues to partition the documents into sections. Sections consist of groups of sentences sharing a common purpose such as "History of Present Illness", "Medications", or "Discharge Instructions". Effective section tagging can reduce the amount of text processed for NLP tasks. This document describes the ClarityNLP section tagger and how it works.

The starting point for the section tagger is the open-source *SecTag* database of J. Denny and colleagues[1].

Source Code

The source code for the section tagger is located in `nlp/algorithms/sec_tag`. The file `sec_tag_db_extract.py` extracts data from the SecTag database, builds the SecTag concept graph (`concept_graph.py`), and generates data files required by the section tagger for its operation. These files are written to the `data` folder. The file `section_tagger.py` contains the code for the section tagger itself.

The section tagger can also run interactively from a command line and process a file of health records in JSON format. The file `sec_tag_file.py` provides a command-line interface to the section tagger. Help can be obtained by running the file with this command: `python3 ./sec_tag_file.py`. This interactive application writes results (input file with tag annotations) to stdout.

SecTag Database

The section tagger requires three input files for its operation, all of which can be found in the `nlp/algorithms/sec_tag/data` folder. These files are `concepts_and_synonyms.txt`, a list of clinical concepts and associated synonyms; `graph.txt`, a list of graph vertices and associated codes for the concept graph, and `normalize.py`, which contains a map of frequently-encountered synonyms and their "normalized" forms[2].

[1] J. Denny, A. Spickard, K. Johnson, N. Peterson, J. Peterson, R. Miller
Evaluation of a Method to Identify and Categorize Section Headers in Clinical Documents
J Am Med Inform Assoc. 16:806-815, 2009.
https://www.vumc.org/cpm/sectag-tagging-clinical-note-section-headers

[2] J. Denny, R. Miller, K. Johnson, A. Spickard
Development and Evaluation of a Clinical Note Section Header Terminology
AMIA Annual Symposium Proceedings 2008, Nov 6:156-160.

Generation of these files requires an installation of the SecTag database. The SecTag SQL files were originally written for MySQL, so that database server will be assumed here. **These files do not need to be generated again unless new concepts and/or synonyms are added to the SecTag database.**

To populate the database, install MySQL and create a root account. Start the MySQL server, log in as root and enter these commands, which creates a user named "sectag" with a password of "sectag":

```
CREATE USER 'sectag'@'localhost' IDENTIFIED BY 'sectag';
CREATE DATABASE SecTag_Terminology;
GRANT ALL ON SecTag_Terminology.* TO 'sectag'@'localhost';
GRANT FILE ON *.* TO 'sectag'@'localhost';
```

The user name and the password can be changed, but the database connection string at the end of `sec_tag_db_extract.py` will need to be updated to match.

After running these commands, log out as the MySQL root user.

Next, download the sec_tag.zip file from the link in[1]. Unzip the file and find `SecTag_Terminology.sql`.

Populate the database as the sectag user with this command, entering the password 'sectag' when prompted:

```
mysql -p -u sectag SecTag_Terminology < SecTag_Terminology.sql
```

The SecTag database name is "SecTag_Terminology". Additional information on the contents of the database can be found in[1] and[2].

Concepts and Synonyms

The section tagger operates by scanning the report text and recognizing synonyms for an underlying set of concepts. The synonyms recognized in the text are mapped to their associated concepts and the document sections are tagged with the concepts. The SecTag database provides an initial set of concepts and synonyms which ClarityNLP expands upon.

For example, concept 158 "history_present_illness" has synonyms "indication", "clinical indication", and "clinical presentation", among others. The synonyms represent the various orthographic forms by which the concept could appear in a clinical note.

The code in `sec_tag_db_extract.py` extracts the concepts and synonyms from the SecTag database; adds new synonyms to the list; adds a few new concepts; corrects various errors occurring in the SecTag database, and writes output to the `nlp/algorithms/sec_tag/data` folder. Run the extraction code with this command:

```
python3 ./sec_tag_db_extract.py
```

Each concept has a "treecode", which is a string consisting of integers separated by periods, such as `6.41.149.234.160.165` (the treecode for the concept "chest_xray"). The numbers encode a path through the concept graph from a small set of general concepts to a much larger set of very specific leaf node concepts. The code 6 represents the concept "objective_data", which is very general and broad in scope. The code 6.41 represents the concept "laboratory_and_radiology_data", which is a form of "objective_data", but more specific. The code 6.41.149 represents the concept "radiographic_studies", which is a more specific form of "laboratory_and_radiology_data". The concepts increase in specificity as the treecodes increase in length. Each node in the concept graph has a unique code that represents a path through the graph from the highest-level concepts to it.

SecTag Errors

There are a few errors in the SecTag database. Two concepts are misspelled. These are concept 127, "principal_diagnosis", misspelled as "principle_diagnosis", and concept 695, "level_of_consciousness", misspelled as

"level_of_cousciousness". ClarityNLP's db extraction code corrects both of these misspellings.

Concept 308, "sleep_habits", has as concept text "sleep_habits,_sleep". The extraction program converts this to just "sleep_habits".

Concept 2921, "preoperative_medications" is missing a treecode. A closely related concept, number 441 "postoperative_medications" has treecode `5.37.106.127` and no children. This concept hierarchy resolves to:

```
patient_history:          5
medications:              5.37
medications_by_situation: 5.37.106
preoperative_medications: 5.37.106.127
```

Using this hierarchy as a guide, the extraction program assigns the treecode `5.37.106.500` to the concept "preoperative_medications".

The final error that the extraction program corrects is for concept 745, "appearance". This entry has an invalid treecode and is an isolated concept at level 10. This strange entry is skipped entirely and is not written to the output files.

Each concept and synonym has a unique integer identifier. The values of these identifiers are all less than 500 for concepts and 6000 for synonyms. The new concepts added by the extraction program begin numbering at 500 and the new synonyms at 6000.

The concepts added by ClarityNLP are:

Concept Name	Treecode
renal_course	5.32.77.79.18.500
preoperative_medications	5.37.106.500
nasopharynx_exam	6.40.139.191.120.500
hypopharynx_exam	6.40.139.191.120.501
xray_ankle	6.41.149.234.160.167.92.500
computed_tomography	6.41.149.234.162.500
cerebral_ct	6.41.149.234.162.500.1
thoracic_ct	6.41.149.234.162.500.2
abdominal_ct	6.41.149.234.162.500.3
renal_and_adrenal_ct	6.41.149.234.162.500.4
extremities_ct	6.41.149.234.162.500.5
nonradiographic_studies	6.41.500
types_of_nonradiographic_studies	6.41.500.1
nonradiographic_contrast_studies	6.41.500.1.1
magnetic_resonance_imaging	6.41.500.1.1.1
cerebral_mri	6.41.500.1.1.1.1
thoracic_mri	6.41.500.1.1.1.2
abdominal_mri	6.41.500.1.1.1.3
renal_and_adrenal_mri	6.41.500.1.1.1.4
extremities_mri	6.41.500.1.1.1.5
magnetic_resonance_angiography	6.41.500.1.1.2
cerebral_mra	6.41.500.1.1.2.1
thoracic_mra	6.41.500.1.1.2.2
abdominal_mra	6.41.500.1.1.2.3
renal_and_adrenal_mra	6.41.500.1.1.2.4
extremities_mra	6.41.500.1.1.2.5

Algorithm

Initialization and Sentence Tokenization

The section tagger begins its operation with an initialization phase in which it loads the data files mentioned above and creates various data structures. One data structure is a mapping of synonyms to concepts, used for fast text lookups. This is a one-to-many mapping since a given synonym can be associated with multiple concepts.

After initialization completes, the section tagger reads the report text and runs the NLTK[3] sentence tokenizer to partition the text into individual sentences. For narrative sections of text the sentence tokenizer performs well. For sections of text containing vital signs, lab results, and extensive numerical data the tokenizer performance is substantially worse. Under these conditions a "sentence" often comprises large chunks of report text spanning multiple sentences and sentence fragments.

Synonym Matching

The section tagger scans each sentence and looks for strings indicating the start of a new section. Clinical note sections tend to be delimited by one or more keywords followed by a termination character. The terminator is usually a colon ":", but dashes and double-dashes also appear as delimeters. The section tagger employs various regular expressions that attempt to match all of these possibilities. The winning match is the longest string of characters among all matches. Any overlapping matches are merged, if possible, prior to deciding the winning match. Each match represents the possible start of a new report section.

For each match, which consists of one or more words followed by a terminator, the section tagger extracts the matching text and performs a series of validity checks on it. Dash-terminated matches are checked to verify that they do not end in the middle of a hyphenated word. They are also checked to ensure that they do not terminate within a hyphenated lab result, such as `SODIUM-135`. Any such matches are discarded. Several other tests are performed as well.

If any matches survive these checks, the terminating characters and possible leading newlines are stripped from the matching text, and any bracketed data (such as anonymized dates) is removed. The remaining text then gets converted to lowercase and searched for concept synonyms and thus candidate headers.

The candidate header discovery processes proceeds first by trying an exact match to the candidate text string. The text itself (after lowercasing) becomes the lookup key for the synonym map built during initialization. If an exact match is found, the associated concept(s) are looked up and inserted into the list of candidate concepts for this portion of report text.

If the exact match fails, the section tagger splits the text into individual words and tries to match the longest sequence of words, if any, to a known synonym. It proceeds to do this by removing words from each end of the word list. It first tries a match anchored to the right, removing words one-by-one from the left. Any matches found are resolved into concepts and added to the candidate concept list. If no matches are found, the section tagger tries again, this time with the matches anchored from the left, and words removed one-by-one from the right. If still no matches are found, the word list is pruned of stop words and the remaining words replaced by their "normalized" forms. The sequence of match attempts repeats on this new word list, first with an exact match, then one anchored right, then one anchored left. If all of these match attempts fail, section tagger gives up and concludes that the text does not represent the start of a new section.

If at least one match attempt succeeds, the synonyms are resolved into concepts via map lookup and returned as candidate concepts for a new section label. If there is only one candidate concept as the result of this process, that concept becomes the header for the next section of text. If two or more candidate concepts remain, the section tagger employs an ambiguity resolution process to decide on the winning concept. The ambiguity resolver uses a concept stack to guide its decisions, which we describe next.

[3] **Natural Language Toolkit**
https://www.nltk.org/

The Concept Stack

The sections in a clinincal note tend to be arranged as flattened hierarchies extending over several consecutive sections. For instance, in a discharge report one might encounter a section labeled GENERAL_EXAM, followed by a section labeled HEAD_AND_NECK_EXAM, which represents a more specific type of general exam. This section could be followed by a section labeled EYE_EXAM, which is an even more specific type of head and neck exam. Although these sections would be listed sequentially in the report, they naturally form a hierarchy of EXAM concepts proceeding from general to specific. Other section groups in the report exhibit the same characteristics.

A data structure for managing hierarchies such as this is a stack. The section tagger manages a "concept stack" as it processes the report text. It uses the stack to identify these natural concept groups, to keep track of the scope of each, and to resolve ambiguities as described in the previous section.

The specificity of a concept is determined by its graph treecode. The longer the treecode, the more specific the concept. Two concepts with identical length treecodes have the same degree of specificity.

Each time the section tagger recognizes a concept C it updates the stack according to this set of empirically-determined rules:

Let T be the concept at the top of the stack.

- If C is a more specific concept than T, push C onto the stack. In other words keep pushing concepts as they get more specific.
- If C has the same specificity as T, pop T from the stack and push C. If two concepts have the same specificity, there is no *a priori* reason to prefer one vs. the other, so take the most recent one.
- If C is more general than T, pop all concepts from the stack that have specificity >= C. In other words, pop all concepts more specific than C, since C could represent the start of a new concept hierarchy.

Thus the section tagger pushes concept C onto the stack if it is more specific than concept T. It pops concepts from the stack until concept T is at the same level of specificity (or less specific) than C. The concepts in the stack represent the full set of open concept scopes at any stage of processing.

Concept Ambiguity Resolution

The section tagger uses the concept stack to select a single concept from a list of candidates, such the candidate concepts produced by the synonym matching process described above. The basic idea is that a concept should be preferred as a section label if it posesses the nearest common ancestor among all concepts in the concept stack. A concept is preferable as a section label if it is "closer" to those in the concept stack than all other candidates. Here the distance metric is the shortest path between the two concept nodes in the concept graph.

The concept ambiguity resolution process proceeds as follows. Let L be a list of concepts and let S be the concept stack. For each concept C in stack S, starting with the concept at the stack top:

- For all candidate concepts in L, find the nearest common ancestor to C.
  - If there is a single ancestor A closer than all others, choose A as the current winner. Save A in the *best_candidates* list. Move one level deeper in the stack and try again.
  - If multiple ancestors are closer than the others, save these as *best_candidates* if they are closer than those already present in *best_candidates*. Move one level deeper in the stack and try again.
  - If all ancestors are at the same level in the concept graph (have the same specificity), there is no clear winner. Move one element deeper in the stack and try again.

This process continues until all elements in the stack have been examined. If one winner among the candidates in L emerges from this procedure, it is declared the winning concept and it is used for the section label.

If there is no single winning concept:

- If there are any *best_candidate* concepts:
  - Select the most general concept from among these as the winner.
  - If all *best_candidate* concepts have the same specificity, select the first of the best candidates as the winner.
- Otherwise, take the most general concept from those in L, if any.
- Otherwise, declare failure for the ambiguity resolution process.

Example

An example may help to clarify all of this. Consider this snippet of text from one of the MIMIC discharge notes:

```
...CV:  The patient's vital signs were routinely monitored, and
was put on vasopressin, norepinephrine and epinephrine during her
stay to maintain appropriate hemodynamics. Pulmonary:  Vital
signs were routinely monitored. She was intubated and sedated
throughout her admission, and her ventilation settings were
adjusted based on ABG values...
```

As the section tagger scans this text it finds a regex match for the text `Pulmonary:`. No additional words match at this point, since this text starts a new sentence. As described above, the section tagger removes the terminating colon and converts the text to lowercase, producing `pulmonary`. It then checks the synonym map for any concepts associated with the text `pulmonary`. It tries an exact match first, which succeeds and produces the following list of candidate concepts and their treecodes (the list L above):

```
L[0]  PULMONARY_COURSE          [5.32.77.87]
L[1]  PULMONARY_FAMILY_HISTORY [5.34.79.103.71]
L[2]  PULMONARY_REVIEW          [5.39.132]
L[3]  PULMONARY_EXAM            [6.40.139.195.128]
L[4]  PULMONARY_PLAN            [13.51.157.296]
```

These are the candidate concepts in list L. The concept stack S at this point is:

```
S[0]  CARDIOVASCULAR_COURSE  [5.32.77.75]
S[1]  HOSPITAL_COURSE        [5.32]
```

How does the section tagger use S to choose the "best" section tag from concepts in L?

To begin, the ambiguity resolution process starts with the concept at the top of the stack, `CARDIOVASCULAR_COURSE`. It proceeds to compute the ancestors shared by this concept and each concept in L. It hopes to find a single most-specific ancestor concept shared between elements of L and S. This is the nearest common ancestor concept for those in L and S.

The nearest common ancestor can be computed from the treecodes. If two treecodes share a common initial digit sequence they have a common ancestor. The treecode of the nearest common ancestor is the **longest shared treecode prefix string**. If two treecodes have no common prefix string they have no common ancestor. The nearest common ancestor for concept A with treecode 6.40.37 and concept B with treecode 6.40.21 is that unique concept with treecode 6.40, since 6.40 is the longest shared prefix string for concepts A and B.

Computing the common ancestors of the concept at the top of the stack, `CARDIOVASCULAR_COURSE [5.32.77.75]`, and each concept in L gives:

```
S[0] & L[0]: [5.32.77]
S[0] & L[1]: [5]
S[0] & L[2]: [5]
```

(continues on next page)

(continued from previous page)

```
S[0] & L[3]: [ ]
S[0] & L[4]: [ ]
```

Concepts `S[0]` and `L[0]` share the longest prefix string. Concepts `L[3]` and `L[4]` share no common ancestor with concept `S[0]`, as the empty brackets indicate. The section tagger declares concept `L[0] PULMONARY_COURSE` to be the winner of this round, since it has the longest shared prefix string with concept `S[0]`, indicating that it is closer to `S[0]` than all other candidate concepts. It then proceeds to the next level in the stack and repeats the procedure, generating these results:

```
S[1] & L[0]: [5.32]
S[1] & L[1]: [5]
S[1] & L[2]: [5]
S[1] & L[3]: [ ]
S[1] & L[4]: [ ]
```

The winner of this round is also `L[0]`, indicating that the node with treecode `5.32` is the nearest common ancestor for concepts `S[1] HOSPITAL_COURSE` and `L[0] PULMONARY_COURSE`. This common ancestor has a shorter treecode than that found in the initial round, indicating that it is located at a greater distance in the concept graph, so the results of this round are discarded.

All elements of the concept stack have been examined at this point, and there is is a single best candidate concept, `L[0] PULMONARY_COURSE`. The section tagger declares this concept to be the winner and labels the section with the tag `PULMONARY_COURSE`. Thefore concept `L[0] PULMONARY_COURSE` shares the nearest common ancestor with those in S, and it is the most appropriate concept with which to label the `Pulmonary:` section.

At this point concept C, which is the most recently-recognized concept, becomes `PULMONARY_COURSE [5.32.77.87]`. The concept T at the top of the stack is `CARDIOVASCULAR_COURSE [5.32.77.75]`. Since concepts C and T have identical treecode lengths, they have the same specificity. Following the stack manipulation rules described above, the section tagger pops the stack and pushes C, which yields this result for the concept stack:

```
S[0]  PULMONARY_COURSE  [5.32.77.87]
S[1]  HOSPITAL_COURSE   [5.32]
```

After these stack adjustments the section tagger resumes scanning and the process continues.

References

ConText

Overview

ConText is based on the algorithm developed by Chapman, et al.[1][2] to determine negation, experiencer and temporality modifiers around clinical concepts. The algorithm uses rules, and text windows (or spans) along with an input concept

[1]

Harkema H, Dowling JN, Thornblade T, Chapman WW
Context: An Algorithm for Determining Negation, Experiencer, and Temporal Status from Clinical Reports
Journal of biomedical informatics. 2009;42(5):839-851.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2757457/

[2]

Chapman WW, Chu D, Dowling JN
ConText: An algorithm for identifying contextual features from clinical text.

to determine the 3 ConText types. Resulting values from ConText can be any of the following, where the bolded item notes the default.

Temporality

- **Recent**
- Historical
- Hypothetical

Experiencer

- **Patient**
- Other

Negation

- **Affirmed**
- Negated
- Possible

Source Code

The source code is found in `nlp/algorithms/context/context.py`.

Concepts

ConText has a pre-defined set of concepts for each ConText type. They can be found at `nlp/algorithms/context/data`. Each ConText keyword has a category which either indicates it as a candidate for a ConText type, a pseudo-candidate (which would be excluded), or a term that indicates a change in the sentence phrase, such as a conjunction (which would close a ConText window).

Algorithm

We have a Python implementation of the ConText algorithm.

References

Lexical Variants

BioNLP Workshop of the Association for Computational Linguistics. June 29, 2007.
http://dl.acm.org/citation.cfm?id=1572408

Overview

ClarityNLP uses the term *lexical variants* to mean either *plurals*, *verb inflections*, or both. Pluralization is a familiar concept and is assumed to be self-explanatory. English verbs have four inflected forms (i.e. a different ending depending on use), which are as follows, using the verb 'walk' as an example:

Description	Inflected Form
bare infinitive (base form)	walk
3rd person singular present	walks
present participle	walking
past tense (preterite)	walked
past participle	walked

Regular English verbs have inflected forms that can be computed from relatively straightforward rules (but there are many exceptions). *Irregular* verbs have inflected forms for the past tense and/or past participle that violate the rules.

ClarityNLP includes a pluralizer and a verb inflector that attempt to compute the plurals and inflected forms of English words. The verb inflector ignores archaic forms and focuses primarily on contemporary American English.

Plurals

The ClarityNLP pluralizer generates plural forms of words and phrases. Several functions are offered depending on whether the part of speech of the term to be pluralized is known. The source code for the pluralizer can be found in `nlp/algorithms/vocabulary/pluralize.py`. The pluralizer is mainly a wrapper around the Python port of Damian Conway's well-known `inflect` module[1]. An error-correction mechanism has also been incorporated to improve the module's performance on medical text.

Inputs

A single string, representing the word or phrase to be pluralized.

Outputs

A list of strings containing all known plural forms for the input.

Functions

The functions provided by the `pluralize` module are (all arguments are strings):

```
plural_noun(noun)
plural_verb(verb)
plural_adj(adjective)
plural(text_string)
```

Use the more specific functions if the part of speech of the input text is known. Use `plural` if nothing is known about the text.

[1] http://users.monash.edu/~damian/papers/extabs/Plurals.html

Verb Inflections

The verb inflector module computes verb inflections from a given verb in base form. The base form of a verb is also known as "plain form", "dictionary form", "bare infinitive form", or as the "principal part" of the verb. Here is a list of some common verbs and their base forms:

Verb	Base Form
running	run
walks	walk
eaten	eat
were	be

It is not possible to unambiguously compute the base form of a verb from an arbitrary inflected form. Observe:

Verb	Possible Base Forms
clad	clad (to cover with material), clothe (to cover with clothes)
cleft	cleave (to split), cleft (to separate important parts of a clause)
fell	fell (to make something fall), fall (to take a tumble)
lay	lay (to set down), lie (to rest on a surface)

The only way to *unambiguously* recover the base form from an arbitrary inflection is to supply additional information such as meaning, pronounciation, or usage.

Lemmatizers attempt to solve this problem, but with decidedly mixed results. Neither the NLTK WordNet lemmatizer nor the Spacy lemmatizer worked reliably enough on this module's test data to allow users to input verbs in arbitrary inflections. Lemmatization is still an area of active NLP research, so these results are not necessarily surprising.

Therefore, for all of these reasons, **the ClarityNLP verb inflector requires the input verb to be provided in base form**.

Source Code

The source code for the verb inflector is located in `nlp/algorithms/vocabulary/verb_inflector.py`. Supporting files in the same directory are `inflection_truth_data.txt`, `irregular_verbs.py`, and the files in the `verb_scraper` directory. The purpose of the supporting files and software will be described below.

Inputs

The entry point to the verb inflector is the `get_inflections` function, which takes a single string as input. The string is a **verb in base form** as described above.

Outputs

The `get_inflections` function returns all inflections for the verb whose base form is given. The inflections are returned as a five-element list, interpreted as follows:

Element	Interpretation
0	[string] the base form of the verb
1	[list] third-person singular present forms
2	[list] present participle forms
3	[list] simple past tense (preterite) forms
4	[list] past participle forms

The lists returned in components 1-4 are all lists of strings. Even if only a single variant exists for one of these components, it is still returned as a single-element list, for consistency.

Example

```
inflections = verb_inflector.get_inflections('outdo')
# returns ['outdo',['outdoes'],['outdoing'],['outdid'],['outdone']]

inflections = verb_inflector.get_inflections('be')
# returns ['be',['is'],['being'],['was','were'],['been']]
```

Algorithms

The verb inflector uses different algorithms for the various inflections. A high-level overview of each algorithm will be presented next. The verb inflector uses a list of 558 irregular verb preterite and past participle forms scraped from Wikipedia and Wiktionary to support its operations.

It should be stated that the rules below have been gleaned from various grammar sources scattered about the Internet. Some grammar sites present subsets of these rules; others present some rules without mentioning any exceptions; and other sites simply present incorrect information. We developed these algorithms iteratively, over a period of time, adjusting for exceptions and violations as we found them. This is still a work in progress.

Algorithm for the Third-Person Singular Present

The third-person singular present can be formed for most verbs, either regular or irregular, by simply adding an `s` character to the end. Some highly irregular verbs such as `be` and a few others are stored in a list of exceptions. If the base form of the verb appears in the exception list, the verb inflector performs a simple lookup and returns the result.

If the base form is not in the exception list, the verb inflector checks to see if it ends in a consonant followed by `y`. If so, the terminating `y` is changed to an `i` and an `es` is added, such as for the verb `try`, which has the third-person singular present form `tries`.

If the base form instead ends in a consonant followed by `o`, an `es` is appended to form the result. An example of such a verb would be `echo`, for which the desired inflection is `echoes`.

If the base form has neither of these endings, the verb inflector checks to see if it ends in a sibilant sound. The sibilant sounds affect the spelling of the third-person singular inflection in the presence of a silent-e ending[2]. The CMU pronouncing dictionary[3] is used to detect the presence of sibilant sounds. The phonemes for these sounds are based on the ARPAbet[4] phonetic transcription codes and appear in the next table:

[2] https://en.wikipedia.org/wiki/English_verbs
[3] http://www.speech.cs.cmu.edu/cgi-bin/cmudict
[4] https://en.wikipedia.org/wiki/ARPABET

Sibilant Sound	Phoneme
voiceless alveolar sibilant	S
voiced alveolar sibilant	Z
voiceless postalveolar fricative	SH
voiced postalveolar fricitave	ZH
voiceless postalveolar affricate	CH
voiced postalveolar affricate	JH

If the base form ends in a sibilant sound and has no silent-e ending, an `es` is appended to form the desired inflection. Otherwise, an `s` is appended to of the base form and returned as the result.

Algorithm for the Present Participle

The verb inflector keeps a dictionary of known exceptions to the rules for forming the present participle. Most of these exceptional verbs are either not found in the CMU pronouncing dictionary, or are modal verbs, auxiliaries, or other irregular forms. Some verbs also have multiple accepted spellings for the present participle, so the verb inflector keeps a list of these as well. If the base form of the given verb appears as an exception, a simple lookup is performed to generate the result.

If the base form of the verb is not a known exception, the verb inflector determines whether the base form ends in `ie`. If it does, the `ie` is changed to `ying` and appended to the base form to generate the result. An example of such a verb is `tie`, which has the form `tying` as the present participle.

Next the verb inflector checks the base form for an `ee`, `oe`, or `ye` ending. If one of these endings is present, the final `e` is retained, and `ing` is appended to the base form and returned as the result.

If the base form ends in vowel-`l`, British spelling tends to double the final `l` before appending `ing`, but American spelling does not. For many verbs both the British and American spellings are common, so the verb inflector generates both forms and returns them as the result. There appears to be one exception to this rule, though. If the vowel preceding the final `l` is an `i`, the rule does not seem to apply (such as for the verb `sail`, whose present participle form is `sailing`, not `sailling`).

If none of these tests succeed, the verb inflector checks for pronounciation- dependent spellings using the CMU pronouncing dictionary. If the base form has a silent-e ending, the final `e` is dropped and `ing` is appended to the base verb to form the result, unless the base form is a known exception to this rule, in which case the final `e` is retained.

The verb inflector next checks for a pronunciation-dependent spelling caused by consonant doubling. The rules for consonant doubling are presented in the next section. The verb inflector doubles the final consonant if necessary, appends `ing`, and returns that as the result.

If none of the tests succeeds, the verb inflector appends `ing` to the base form and returns that as the result.

Algorithm for Consonant Doubling

If the base form of the verb ends in `c`, a `k` should generally be appended prior to the inflection ending. There are a few exceptions to this rule that the verb inflector checks for.

If the base form of the verb ends in two vowels followed by a consonant, the rule is generally to not double the final consonant. One exception to this rule is if the first vowel is a `u` preceded by `q`. In this case the `u` is pronounced like a `w`, so the `qu` acts as if it were actually `qw`. This gives the word an effective consonant-vowel-consonant ending, in which case the final consonant is doubled. An example of this would be the verb `equip`, which requires a doubled `p` for inflection (`equipping`, `equipped`, etc.).

If the base form of the verb has a vowel-consonant ending, and if the consonant is not a silent-t, then the final consonant is doubled for single syllable verbs. If the final syllable is stressed, the final consonant is also doubled. Otherwise the final consonant is not doubled prior to inflection.

Algorithm for the Simple Past Tense

If the verb is irregular, its past tense inflection cannot be predicted, so the verb inflector simply looks up the past tense form in a dict and returns the result. A lookup is also performed for a small list of regular verbs that are either known exceptions to the rules, or which have multiple accepted spellings for the past tense forms.

If the verb is regular and not in the list of exceptions, the verb inflector checks the base form for an `e` ending. If the verb ends in `e`, a `d` is appended and returned as the result.

If the base form instead ends in a consonant followed by `y`, the `y` is changed to `i` and `ed` is appended and returned as the result.

If the base form ends in a vowel followed by `l`, both the American and British spellings are returned, as described above for the present participle. The British spelling appends `led` to the base form, while the American spelling only appends `ed`.

If the final consonant requires doubling, the verb inflector appends the proper consonant followed by `ed` and returns that as the result.

Otherwise, `ed` is appended to the base form and returned as the result.

Algorithm for the Past Participle

The past participle for irregular verbs is obtained by simple lookup. The past participle for a small number of regular verbs with multiple accepted spellings is also obained via lookup. Otherwise, the past participle for regular verbs is equivalent to the simple past tense form.

Testing the Verb Inflector

The file `verb_inflector.py` includes 114 test cases that can be run via the `--selftest` command line option. A more extensive set of 1364 verbs and all inflected forms can be found in the file `inflection_truth_data.txt`. This list consists of the unique verbs found in two sets: the set of irregular English verbs scraped from Wikipedia[5], and the set of the 1000 most common English verbs scraped from poetrysoup.com[6]. The verb_inflector will read the file, compute all inflections for each verb, and compare with the data taken from the file using this command:

```
python3 ./verb_inflector.py -f inflection_truth_data.txt
```

The code for scraping the verbs and generating the truth data file can be found in the `verb_scraper` folder.

To generate the truth data file, change directories to the `verb_scraper` folder and run this command:

```
python3 ./scrape_verbs.py
```

Two output files will be generated:

- `verb_list.txt`, a list of the unique verbs found
- `irregular_verbs.py`, data structures imported by the verb inflector

[5] https://en.wikipedia.org/wiki/List_of_English_irregular_verbs
[6] https://www.poetrysoup.com/common_words/common_verbs.aspx

In addition to scraping verb data, this code also corrects for some inconsistencies found between Wikipedia and the Wiktionary entries for each verb.

Copy `irregular_verbs.py` to the folder that contains `verb_inflector.py`, which should be the parent of the `verb_scraper` folder.

Next, scrape the inflection truth data from Wiktionary for each verb in `verb_list.txt`:

```
python3 ./scrape_inflection_data.py
```

This code loads the verb list, constructs the Wiktionary URL for each verb in the list, scrapes the inflection data, corrects further inconsistencies, and writes the output file `raw_inflection_data.txt`. Progress updates appear on the screen as the run progresses.

Finally, generate the truth data file with this command:

```
python3 ./process_scraped_inflection_data.py
```

References

Sentence Tokenization

Overview

Sentence tokenization is the process of splitting text into individual sentences. For literature, journalism, and formal documents the tokenization algorithms built in to spaCy perform well, since the tokenizer is trained on a corpus of formal English text. The sentence tokenizer performs less well for electronic health records featuring abbreviations, medical terms, spatial measurements, and other forms not found in standard written English.

ClarityNLP attempts to improve the results of the sentence tokenizer for electronic health records. It does this by looking for the types of textual constructs that confuse the tokenizer and replacing them with single words. The sentence tokenizer will not split an individual word, so the offending text, in replacement form, is preserved intact during the tokenization process. After generating the individual sentences, the reverse substitutions are made, which restores original text in a set of improved sentences. ClarityNLP also performs additional fixups of the sentences to further improve the results. This document will describe the process and illustrate with an example.

Source Code

The source code for the sentence tokenizer is located in `nlp/algorithms/segmentation/segmentation.py`, with supporting code in `nlp/algorithms/segmentation/segmentation_helper.py`.

Inputs

The entry point to the sentence tokenizer is the `parse_sentences` method of the `Segmentation` class. This function takes a single argument, the text string to be split into sentences.

Outputs

The `parse_sentences` method returns a list of strings, which are the individual sentences.

Example

```
seg_obj = Segmentation()
sentence_list = seg_obj.parse_sentences(my_text)
```

Algorithm

The improvement process proceeds through several stages, which are:

1. Perform cleanup operations on the report text.
2. Perform textual substitutions.
3. Run the spaCy sentence tokenizer on the cleaned, substituted text.
4. Find and split two consecutive sentences with no space after the period.
5. Undo the substitutions.
6. Perform additional sentence fixups for some easily-detectable errors.
7. Place all-caps section headers in their own sentence.
8. Scan the resulting sentences and delete any remaining errors.

Additional explanations for some of these items are provided below.

Text Cleanup

The text cleanup process first searches the report text for cut-and-paste section headers found between `(Over)` and `(Cont)` tokens. These headers are often inserted directly into a sentence, producing a confusing result. Here is an example:

"There are two subcentimeter right renal hypodensities, 1 in\n (Over)\n\n [**2728-6-8**] 5:24 PM\n CT CHEST W/CONTRAST; CT ABD & PELVIS W & W/O CONTRAST, ADDL SECTIONSClip # [**Telephone/Fax (1) 103840**]\n Reason: Evaluate for metastasis/lymphadenopathy related to ? GI [**Country **]\n Admitting Diagnosis: UPPER GI BLEED\n Contrast: OMNIPAQUE Amt: 130\n _____\n FINAL REPORT\n (Cont)\n the upper pole and 1 in the lower pole, both of which are too small to\n characterize."

By looking at this text closely, you can see how the `(Over)..(Cont)` section has been pasted into this sentence:

"There are two subcentimeter right renal hypodensities, 1 in the upper pole and 1 in the lower pole, both of which are too small to\n characterize."

The meaning of this passage is not obvious to a human observer on first inspection, and it completely confuses a sentence tokenizer trained on standard English text.

ClarityNLP finds these pasted report headers and removes them.

The next step in the cleanup process is the identification of numbered lists. The numbers are removed and the narrative descriptions following the numbers are retained.

As is visible in the pasted section header example above, electronic health records often contain long runs of dashes, asterisks, or other symbols. These strings are used to delimit sections in the report, but they are of no use for machine interpretation, so ClarityNLP searches for and removes such strings.

Finally, ClarityNLP locates any instances of repeated whitespace (which includes spaces, newlines, and tabs) and replaces them with a single space.

Textual Substitutions

ClarityNLP performs several different types of textual substitution prior to sentence tokenization. All of these constructs can potentially cause problems:

Construct	Example
Abbreviations	.H/O, Sust. Rel., w/
Vital Signs	VS T97.3 P84 BP120/56 RR16 O2Sat98 2LNC
Capitalized Header	INDICATION:
Anonymizations	[**2728-6-8**], [**Telephone/Fax (1) 103840**]
Contrast Agents	Conrast: OMNIPAQUE Amt: 130
Field of View	Field of view: 40
Size Measurement	3.1 x 4.2 mm
Dispensing Info	Protonix 40 mg p.o. q. day.
Gender	Sex: M

ClarityNLP uses regular expressions to find instances of these constructs. Wherever they occur they are replaced with single-word tokens such as "ANON000", "ABBREV001", "MEAS002", etc. Replacements of each type are numbered sequentially. The sentence tokenizer sees these replacements as single words, and it preserves them unchanged through the tokenization process. These replacements can be easily searched for and replaced in the resulting sentences.

Split Consecutive Sentences

The punctuation in electronic health records does not always follow standard forms. Sometimes consecutive sentences in a report have a missing space after the period of the first sentence, which can cause the sentence tokenizer to treat both sentences together as a single run-on sentence. ClarityNLP detects these occurrences and separates the sentences. It also avoids separating valid abbreviations such as *C.Diff.*, *G.Jones*, etc.

Perform Additional Sentence Fixups

Sometimes the sentence tokenizer generates sentences that begin with a punctuation character such as `:` or `,`. ClarityNLP looks for such occurrences and moves the punctuation to the end of the preceding sentence.

Delete Remaining Errors

ClarityNLP scans the resulting set of sentences and takes these actions:

- deletes any remaining list numbering
- deletes any sentences consisting only of list numbering
- removes any sentences that consist only of '#1', '#2', etc.
- removes any sentences consisting entirely of nonalphanumeric symbols
- concatenates sentences that incorrectly split an age in years
- concatenates sentences that split the subject of a measurement from the measurement

Example

Here is a before and after example illustrating several of the tokenization problems discussed above. The data is taken from one of the reports in the MIMIC data set.

BEFORE: Each numbered string below is a sentence that emerges from the sentence tokenizer without ClarityNLP's additional processing. Note that the anonymized date and name tokens `[** ... **]` are broken apart, as are numbered lists, drug dispensing information, vital signs, etc. You can see how the sentence tokenizer performs better for the narrative sections, but the abbreviations and other nonstandard forms confuse it and cause errors:

```
[  0]        Admission Date:  [
[  1]        **3104-4-26
[  2]        **]     Discharge Date:  [**3104-4-28
[  3]        **]


Service:  CARDIAC CA

CHIEF COMPLAINT:   Dyspnea on exertion.

HISTORY OF PRESENT ILLNESS:
[  4]        This is a 78 year old male with
hypertension and hyperlipidemia who was in his usual state of health until two weeks␣
↪prior to admission when he noted increasing shortness of breath on exertion,␣
↪especially with stairs.
[  5]        Since that time, the patient reports decreased exercise tolerance but␣
↪denied any orthopnea, paroxysmal nocturnal dyspnea, or lower extremity swelling.
[  6]        He denies any dizziness or lightheadedness.
[  7]        He was seen in Dr.
[  8]        [**Last Name (STitle) 23973*
[  9]        *]
[ 10]        [**Name (STitle) 23974
[ 11]        *
[ 12]        *]
[ 13]        Clinic the day of admission and was found to have
high grade infra-nodal heart block and was sent to the Emergency Room.
[ 14]        A central line was placed with temporary
pacing wire placed overnight.
[ 15]        PAST MEDICAL HISTORY:
1.  Hypertension.
[ 16]        2.
[ 17]        Hyperlipidemia.
[ 18]        3.
[ 19]        Exercise thallium stress test in [**3100*
[ 20]        *] showed a small
basal inferior fixed defect.
[ 21]        4.
[ 22]        Mild asthma.
[ 23]        5.
[ 24]        Hemorrhoids.
[ 25]        6.
[ 26]        Colonic polyps.
[ 27]        7.
[ 28]        Left bundle branch block since [
[ 29]        **3098-10-8**].
8.
[ 30]        Bilateral hernia repairs.
```

(continues on next page)

(continued from previous page)

```
[ 31]        ALLERGIES:
[ 32]        He has no known drug allergies.
[ 33]        MEDICATIONS:
[ 34]        1.  Hydrochlorothiazide 12.5 mg
[ 35]        p.o.
[ 36]        q. day.
[ 37]        2.
[ 38]        Lipitor 40 mg
[ 39]        p.o.
[ 40]        q.
[ 41]        h.s.
[ 42]        3.
[ 43]        Enalapril 20
[ 44]        mg p.o. twice a day.
[ 45]        4.
[ 46]        Cardizem 180 mg p.o.
[ 47]        q. day.
[ 48]        5.
[ 49]        Aspirin 81 mg
[ 50]        p.o.
[ 51]        q. day.
[ 52]        SOCIAL HISTORY:
[ 53]        He has a remote tobacco history; quit over
25 years ago.
[ 54]        He has a remote alcohol history; quit over 17
years ago.
[ 55]        FAMILY HISTORY:   Family history of stroke but denies any
family history of coronary artery disease or malignancy.
[ 56]        PHYSICAL EXAMINATION:   Temperature
[ 57]        is 98.0 F.; heart rate 35
to 45; blood pressure 161/32; respiratory rate 19; 98% on room air.
[ 58]        In no acute distress.
[ 59]        Pupils were reactive to light; the left was 3 millimeters to 2␣
↪millimeters; on the right it was 2 millimeters to 1 millimeters.
[ 60]        Extraocular movements intact.
[ 61]        Mucous membranes were moist.
[ 62]        Jugular venous pressure at about 7 centimeters.
[ 63]        Lungs were clear to auscultation bilaterally.
[ 64]        He is bradycardic with normal S1 and S2 with I/VI systolic murmur at the␣
↪apex.
[ 65]        His abdomen was soft, nontender, nondistended, with normoactive bowel␣
↪sounds.
[ 66]        No edema.
[ 67]        In his extremities he had two plus dorsalis pedis bilaterally.
[ 68]        LABORATORY:
[ 69]        EKG showed sinus with atrial rate of 70, 2:1
heart block with ventricular rate of 35 and an old left bundle branch block.
[ 70]        White blood cell count 11.3, hematocrit 34.6, platelets 298.
[ 71]        Sodium 140, potassium 4.1, chloride 102, bicarbonate 25, BUN
26, creatinine 1.3, glucose 129.
[ 72]        CK 96.
[ 73]        Troponin less than
0.3.

Echocardiogram in [
[ 74]        **3103-2-6
[ 75]        **] showed a large left atrium,
```

(continues on next page)

(continued from previous page)

```
ejection fraction 60 to 65% with mild symmetric left ventricular hypertrophy, trace␣
↪aortic regurgitation, mild mitral regurgitation.
[ 76]         INR was 1.2, PTT 22.7.
[ 77]         Total cholesterol in [**3104-2-6**]
showed total cholesterol of 161, LDL 89, HDL of 35, triglycerides of 184.
[ 78]         Urinalysis was negative.
[ 79]         Chest x-ray was negative.
[ 80]         HOSPITAL COURSE:
[ 81]         The patient remained stable in the
hospital.
[ 82]         He underwent electrophysiology study and pacemaker placement.
[ 83]         He remained stable and asymptomatic.
[ 84]         He was then discharged home.
[ 85]         DISCHARGE
[ 86]         INSTRUCTIONS:
[ 87]         1.
[ 88]         Not to lift anything heavier than ten pounds for two
weeks with the left arm.
[ 89]         2.
[ 90]         He was asked to call his cardiologist with any fatigue or
shortness of breath.
[ 91]         3.
[ 92]         He was to follow-up in Device Clinic in one week.
[ 93]         4.
[ 94]         He was to follow-up with his cardiologist in two to three
weeks.
[ 95]         DISCHARGE DIAGNOSES:
[ 96]         1.
[ 97]         Complete heart block.
[ 98]         MAJOR
[ 99]         INTERVENTIONS:
[100]         1.
[101]         Transvenous pacer wire placement on [**4-26
[102]         **].
[103]         2.
[104]         Pacemaker placement on [
[105]         **4-27
[106]         **].
[107]         CONDITION ON DISCHARGE:   Stable.

DISCHARGE
[108]         MEDICATIONS:
[109]         1.
[110]         Enalapril 20
[111]         mg p.o. twice a day.
[112]         2.
[113]         Hydrochlorothiazide 12.5 mg p.o.
[114]         q. day.
[115]         3.
[116]         Lipitor 40 mg
[117]         p.o.
[118]         q.
[119]         h.s.
[120]         4.
[121]         Percocet p.r.n.
5.
[122]         Keflex 500 mg p.o.
```

(continues on next page)

(continued from previous page)

```
[123]         q. six hours for three days.
[124]         6.
[125]         Ativan 1 mg p.o.
[126]         q.
[127]         h.s.
[128]         as needed.
[129]         7.
[130]         Diltiazem 180 mg p.o.
[131]         q. day.
[132]         [**First Name8 (NamePattern2)
[133]         *
[134]         *]
[135]         [
[136]         **First Name8 (NamePattern2) 1682
[137]         *
[138]         *]
[139]         [**Name8 (MD)
[140]         *
[141]         *], M.D.  [**MD Number(1) 1683
[142]         **]

Dictated By:[**Name8 (MD) 5378
[143]         **]

MEDQUIST36

D:
[144]         [**3104-4-29
[145]         **]  11:19
T:  [
[146]         *
[147]         *3104-5-2**]  21:56
JOB#:  [
[148]         **Job Number 23975**]
```

AFTER: Here is the same report after ClarityNLP does the cleanup, substitutions, and additional processing described above:

```
[  0]         Admission Date: [**3104-4-26**] Discharge Date: [**3104-4-28**] Service:
[  1]         CARDIAC CA CHIEF COMPLAINT:
[  2]         Dyspnea on exertion.
[  3]         HISTORY OF PRESENT ILLNESS:
[  4]         This is a 78 year old male with hypertension and hyperlipidemia who was␣
↪in his usual state of health until two weeks prior to admission when he noted␣
↪increasing shortness of breath on exertion, especially with stairs.
[  5]         Since that time, the patient reports decreased exercise tolerance but␣
↪denied any orthopnea, paroxysmal nocturnal dyspnea, or lower extremity swelling.
[  6]         He denies any dizziness or lightheadedness.
[  7]         He was seen in Dr. [**Last Name (STitle) 23973**] [**Name (STitle)␣
↪23974**] Clinic the day of admission and was found to have high grade infra-nodal␣
↪heart block and was sent to the Emergency Room.
[  8]         A central line was placed with temporary pacing wire placed overnight.
[  9]         PAST MEDICAL HISTORY:
[ 10]         Hypertension.
[ 11]         Hyperlipidemia.
[ 12]         Exercise thallium stress test in [**3100**] showed a small basal␣
↪inferior fixed defect.
```

(continues on next page)

(continued from previous page)

```
[ 13]        Mild asthma.
[ 14]        Hemorrhoids.
[ 15]        Colonic polyps.
[ 16]        Left bundle branch block since [**3098-10-8**].
[ 17]        Bilateral hernia repairs.
[ 18]        ALLERGIES:
[ 19]        He has no known drug allergies.
[ 20]        MEDICATIONS:
[ 21]        Hydrochlorothiazide 12.5 mg p.o. q. day.
[ 22]        Lipitor 40 mg p.o. q. h.s.
[ 23]        Enalapril 20 mg p.o. twice a day.
[ 24]        Cardizem 180 mg p.o. q. day.
[ 25]        Aspirin 81 mg p.o. q. day.
[ 26]        SOCIAL HISTORY:
[ 27]        He has a remote tobacco history; quit over 25 years ago.
[ 28]        He has a remote alcohol history; quit over 17 years ago.
[ 29]        FAMILY HISTORY:
[ 30]        Family history of stroke but denies any family history of coronary␣
↪artery disease or malignancy.
[ 31]        PHYSICAL EXAMINATION:
[ 32]        Temperature is 98.0 F.; heart rate 35 to 45; blood pressure 161/32;␣
↪respiratory rate 19; 98% on room air.
[ 33]        In no acute distress.
[ 34]        Pupils were reactive to light; the left was 3 millimeters to 2␣
↪millimeters; on the right it was 2 millimeters to 1 millimeters.
[ 35]        Extraocular movements intact.
[ 36]        Mucous membranes were moist.
[ 37]        Jugular venous pressure at about 7 centimeters.
[ 38]        Lungs were clear to auscultation bilaterally.
[ 39]        He is bradycardic with normal S1 and S2 with I/VI systolic murmur at the␣
↪apex.
[ 40]        His abdomen was soft, nontender, nondistended, with normoactive bowel␣
↪sounds.
[ 41]        No edema.
[ 42]        In his extremities he had two plus dorsalis pedis bilaterally.
[ 43]        LABORATORY:
[ 44]        EKG showed sinus with atrial rate of 70, 2:1 heart block with␣
↪ventricular rate of 35 and an old left bundle branch block.
[ 45]        White blood cell count 11.3, hematocrit 34.6, platelets Sodium 140,␣
↪potassium 4.1, chloride 102, bicarbonate 25, BUN 26, creatinine 1.3, glucose 129.
[ 46]        CK Troponin less than 0.
[ 47]        Echocardiogram in [**3103-2-6**] showed a large left atrium, ejection␣
↪fraction 60 to 65% with mild symmetric left ventricular hypertrophy, trace aortic␣
↪regurgitation, mild mitral regurgitation.
[ 48]        INR was 1.2, PTT 22.
[ 49]        Total cholesterol in [**3104-2-6**] showed total cholesterol of 161, LDL␣
↪89, HDL of 35, triglycerides of Urinalysis was negative.
[ 50]        Chest x-ray was negative.
[ 51]        HOSPITAL COURSE:
[ 52]        The patient remained stable in the hospital.
[ 53]        He underwent electrophysiology study and pacemaker placement.
[ 54]        He remained stable and asymptomatic.
[ 55]        He was then discharged home.
[ 56]        DISCHARGE INSTRUCTIONS:
[ 57]        Not to lift anything heavier than ten pounds for two weeks with the left␣
↪arm.
[ 58]        He was asked to call his cardiologist with any fatigue or shortness of␣
↪breath.
```

(continues on next page)

(continued from previous page)

```
[ 59]        He was to follow-up in Device Clinic in one week.
[ 60]        He was to follow-up with his cardiologist in two to three weeks.
[ 61]        DISCHARGE DIAGNOSES:
[ 62]        Complete heart block.
[ 63]        MAJOR INTERVENTIONS:
[ 64]        Transvenous pacer wire placement on [**4-26**].
[ 65]        Pacemaker placement on [**4-27**].
[ 66]        CONDITION ON DISCHARGE:
[ 67]        Stable.
[ 68]        DISCHARGE MEDICATIONS:
[ 69]        Enalapril 20 mg p.o. twice a day.
[ 70]        Hydrochlorothiazide 12.5 mg p.o. q. day.
[ 71]        Lipitor 40 mg p.o. q. h.s.
[ 72]        Percocet p.r.n.
[ 73]        Keflex 500 mg p.o. q. six hours for three days.
[ 74]        Ativan 1 mg p.o. q. h.s. as needed.
[ 75]        Diltiazem 180 mg p.o. q. day.
[ 76]        [**First Name8 (NamePattern2) **]
[ 77]        [**First Name8 (NamePattern2) 1682**]
[ 78]        [**Name8 (MD) **], M.D.
[ 79]        [**MD Number(1) 1683**] Dictated By:[**Name8 (MD) 5378**] MEDQUIST36
[ 80]        D:
[ 81]        [**3104-4-29**] 11:19
[ 82]        T:
[ 83]        [**3104-5-2**]
[ 84]        21:56
[ 85]        JOB#:
[ 86]        [**Job Number 23975**]
```

Note that there are fewer sentences overall, and that each sentence has a much more standard form than those in the 'before' panel above. The drug dispensing instructions have been been corrected, the list numbering has been removed, and the patient temperature that was split across sentences 56 and 57 has been restored (new sentence 32).

Command Line Interface

The sentence tokenizer has a command line interface that can be used for inspecting the generated sentences. The input data must be a JSON-formatted file with the proper ClarityNLP fields. This file can be produced by querying SOLR for the reports of interest and dumping the results as a JSON-formatted file. The sentence tokenization module will read the input file, split the text into sentences as described above, and write the results to stdout. Help for the command line interface can be obtained by running this command from the `nlp/algorithms/segmentation` folder:

```
python3 ./segmentation.py --help
```

Some examples:

To tokenize all reports in myreports.json and print each sentence to stdout:

```
python3 ./segmentation.py --file /path/to/myreports.json
```

To tokenize only the first 10 reports (indices begin with 0):

```
python3 ./segmentation.py --file myreports.json --end 9``
```

To tokenize reports 115 through 134 inclusive, and to also show the report text after cleanup and token substitution (i.e. the actual input to the spaCy sentence tokenizer):

```
python3 ./segmentation.py --file myreports.json --start 115 --end 134 --debug
```

Term-Frequency Matrix Preprocessor

Overview

Term-frequency matrices feature prominently in text processing and topic modeling algorithms. In these problems one typically starts with a set of documents and a list of words (the *dictionary*). A term-frequency matrix is constructed from the dictionary and the document set by counting the number of occurrences of each dictionary word in each document. If the rows of the matrix index the words and the columns index the documents, the matrix element at coordinates *(r, c)* represents the number of occurrences of dictionary word *r* in document *c*. Thus each entry of the matrix is either zero or a positive integer.

Construction of such a matrix is conceptually simple, but problems can arise if the matrix contains duplicate rows or columns. The presence of duplicate columns means that the documents at those indices are *identical* with respect to the given dictionary. The linear algebra algorithms underlying many text processing and information retrieval tasks can exhibit instability or extremely slow convergence if duplicates are present. Mathematically, a term-frequency matrix with duplicate columns has a rank that is numerically less than the column count. Under such conditions it is advantageous to remove the duplicated columns (and/or rows) and work with a smaller, fuller-rank matrix.

The *ClarityNLP matrix preprocessor* is a command-line tool that scans a term-frequency matrix looking for duplicate rows and columns. If it finds any duplicates it prunes them and keeps only one row or column from each set of duplicates. After pruning it scans the matrix again, since removal of rows or columns could create further duplicates. This process of scanning and checking for duplicates proceeds iteratively until either a stable matrix is achieved or nothing is left (a rare occurrence, mainly for ill-posed problems). The resulting matrix is written to disk, along with the surviving row and column index lists.

Source Code

The source code for the matrix preprocessor is located in `nlp/algorithms/matrix_preprocessor`. The code is written in C++ with a python driver `preprocess.py`.

Building the Code

A C++ compiler is required to build the matrix preprocessor.

On Linux systems, use your package manager to install the `build-essential` package, which should contain the Gnu C++ compiler and other tools needed to build C++ code. After installation, run the command `g++ --version`, which should print out the version string for the Gnu C++ compiler. If this command produces a `command not found` error, then use your package manager to explicitly install the `g++` package.

On MacOSX, install the xcode command-line tools with this command: `xcode-select --install`. After installation run the command `clang++ --version`, which should generate a version string for the clang C++ compiler.

After verifying that the C++ compiler works, build the matrix preprocessor code with these commands:

```
cd nlp/algorithms/matrix_preprocessor
make
```

The build process should run to completion with no errors, after which these binaries should be present in the `build/bin` folder: `libpreprocess.a`, `preprocessor`, and `test_preprocessor`.

Inputs

The matrix preprocessor requires a single input file. The input file must be in MatrixMarket format, a popular and efficient format for sparse matrices.

Python supports the MatrixMarket format via the `scipy` module and the functions `scipy.io.mmwrite` and `scipy.io.mmread`.

Input Options

The matrix preprocessor supports the following set of command line options. All are optional except for `--infile`, which specifies the file containing the term-frequency matrix to be processed:

Option	Argument	Explanation
`-i, --infile`	string	path to input file, MatrixMarket format
`-r, --min_docs_per_term`	integer	min number of docs per dictionary term, default 3
`-c, --min_terms_per_doc`	integer	min number of dictionary terms per doc, default 5
`-p, --precision`	integer	precision of values in output file, default 4 digits (valid only if `--weights` flag is present)
`-w, --weights`	none	if present, generate TF-IDF weights for entries and output a floating point term-document matrix
`-b, --boolean`	none	if present, enable boolean mode, in which nonzero values in the input matrix are set to 1
`-h, --help`	none	print user help to stdout
`-v, --version`	none	print version information to stdout

The `--min_docs_per_term` option is the cutoff value for pruning rows. Any dictionary term that appears in fewer than this many documents will be pruned. In other words, a row of the input matrix will be pruned if its row sum is less than this value.

Similarly, the `--min_terms_per_doc` option is the cutoff value for pruning columns. Any document that contains fewer than this many dictionary words will be pruned. In other words, a column of the input matrix will be pruned if its column sum is less than this value.

Outputs

The matrix preprocessor generates three output files.

One file, `reduced_dictionary_indices.txt`, is a list of row indices from the original matrix that survived the pruning process. Another file, `reduced_document_indices.txt`, contains a list of original document indices that survived the pruning process.

The third file, in MatrixMarket format, is the pruned matrix. The contents and name of this file depend on whether the `--weights` flag was used for the run.

If the `--weights` flag was absent, the output is another term-frequency matrix in MatrixMarket format. The output file name is `reduced_matrix_tf.mtx` and it contains nonnegative integer entries.

If the `--weights` flag was present, the output is a term-document matrix containing TF_IDF weights for the entries. In this case the output file name is `reduced_matrix.mtx` and it contains floating point entries. The precision of each entry is set by the `--precision` flag.

All output files are written to the current directory.

Examples

1. Prune duplicate rows/columns from the input term-frequency matrix. Write pruned matrix to `reduced_matrix_tf.mtx`; generate the two index files as well:

```
python3 ./preprocess.py --infile /path/to/mymatrix.mtx
```

2. Same as in example 1, but generate an output term-document matrix containing TF-IDF weights. Write result matrix to `reduced_matrix.mtx`; generate the two index files also:

```
python3 ./preprocess.py --infile /path/to/mymatrix.mtx --weights
```

3. Same as 2, but require a mininim row sum of 6 and a mininum column sum of 8 in the pruned term-frequency matrix. Compute TF-IDF weights and output a floating point term-document matrix.

```
python ./preprocess.py -i /path/to/mymatrix.mtx -r 6 -c 8 -w
```

Important Note

The matrix preprocessor was designed for sparse matrices. The term-frequency matrices that occur in typical text processing problems are extremely sparse, with occupancies of only a few percent. Dense matrices should be handled with different techniques.

Task Algorithms

Term Finder

The most basic algorithm, which uses regular expressions to identify terms. In addition, the algorithm will return section, negation, experiencer and temporality. Runs the *ConText* and *section tagging* algorithms.

Provider Assertion

An extension of Term Finder, which uses regular expressions to identify terms. In addition, the algorithm will return section, negation, experiencer and temporality from *ConText*, but will filter them such that the follow conditions are met:

- **Negation**: Affirmed
- **Experiencer**: Patient
- **Temporality**: Historical OR Recent

Finding Date Expressions

Overview

ClarityNLP includes a module that locates date expressions in clinical text. By 'date expression' we mean a string such as `July 20, 1969`, `7.20.69`, or something similar. The `DateFinder` module scans sentences for date expressions, extracts them, and generates output in JSON format.

Source Code

The source code for the date finder module is located in `nlp/algorithms/finder/date_finder.py`.

Inputs

A single string, the sentence to be scanned for date expressions.

Outputs

A JSON array containing these fields for each date expression found:

Field Name	Explanation
text	string, text of the complete date expression
start	integer, offset of the first character in the matching text
end	integer, offset of the final character in the matching text plus 1
year	integer year
month	integer month (Jan=1, Feb=2, . . . , Dec=12)
day	integer day of the month [1, 31]

All JSON results contain an identical number of fields. Any fields that are not valid for a given date expression will have a value of EMPTY_FIELD and should be ignored.

Algorithm

ClarityNLP uses a set of regular expressions to recognize date expressions. The date_finder module scans a sentence with each date-finding regex and keeps track of any matches. If any matches overlap, an overlap resolution process is used to select a winniner. Each winning match is converted to a `DateValue` namedtuple. This object is defined at the top of the source code module and can be imported by other Python code. Each namedtuple is appended to a list as the sentence is scanned. After scanning completes, the list of `DateValue` namedtuples is converted to JSON and returned to the caller.

Date Expression Formats

Using notation similar to that used by the PHP date reference, we define the following quantities:

Shorthand	Meaning
dd	one or two-digit day of the month with optional suffix (7th, 22nd, etc.)
DD	two-digit day of the month
m	textual name of the month
M	textual month abbreviation
mm	one or two-digit numerical month
MM	two-digit month
y	two or four-digit year
yy	two-digit year
YYYY	four-digit year
?	optional

With these definitions, the date expression formats that ClarityNLP recognizes are (using the date of the first Moon landing for illustration):

Date Expression Format	Examples
YYYYMMDD	19690720
[-+]?YYYY-MM-DD	+1969-07-20
YYYY/MM/DD	1969/07/20
YY-MM-DD	69-07-20
YYYY-MM-DDTHH:MM:SS(.ffffff)?	
(here MM:SS means minutes and seconds)	
mm/dd/YYYY	07/20/1969
YYYY/mm/dd	1969/7/20, 1969/07/20
dd-mm-YYYY, dd.mm.YYYY	20-07-1969, 20.7.1969
y-mm-dd	1969-7-20, 1969-07-20, 69-7-20
dd.mm.yy	20.7.69, 20.07.69
dd-m-y, ddmy, dd m y	20-JULY-69, 20JULY69, 20 July 1969
m-dd-y, m.dd.y, mddy, m dd, y	20-July 1969, 20JULY1969, 20 July, 1969
M-DD-y	Jul-20-1969, Jul-20-69
y-M-DD	69-Jul-20, 1969-Jul-20
mm/dd	7/20, 07/20
m-dd, m.dd, m dd	July 20, July 20th, July-20
dd-m, dd.m, dd m	20-July, 20.July, 20 July
YYYY-mm	1969-07, 1969-7
m-YYYY, m.YYYY, m YYYY	July-1969, July.1969, July 1969
YYYY-m, YYYY.m, YYYY m	1969-July, 1969.July, 1969 July
YYYY	1969
m	July

Finding Time Expressions

Overview

ClarityNLP includes a module that locates time expressions in clinical text. By 'time expression' we mean a string such as `9:41 AM`, `05:12:24.12345`, or something similar. The `TimeFinder` module scans sentences for time expressions, extracts them, and generates output in JSON format.

Source Code

The source code for the time finder module is located in `nlp/algorithms/finder/time_finder.py`.

Inputs

A single string, the sentence to be scanned for time expressions.

Outputs

A JSON array containing these fields for each time expression found:

Field Name	Explanation
text	string, text of the complete time expression
start	integer, offset of the first character in the matching text
end	integer, offset of the final character in the matching text plus 1
hours	integer hours
minutes	integer minutes
seconds	integer seconds
fractional_seconds	string, contains digits after decimal point, including any leading zeros
am_pm	string, either STR_AM or STR_PM (see values below)
timezone	string, timezone code
gmt_delta_sign	sign of the UTC offset, either '+' or '-'
gmt_delta_hours	integer, UTC hour offset
gmt_delta_minutes	integer, UTC minute offset

All JSON results contain an identical number of fields. Any fields that are not valid for a given time expression will have a value of EMPTY_FIELD and should be ignored.

Algorithm

ClarityNLP uses a set of regular expressions to recognize time expressions. The time_finder module scans a sentence with each time-finding regex and keeps track of any matches. If any matches overlap, an overlap resolution process is used to select a winniner. Each winning match is converted to a `TimeValue` namedtuple. This object is defined at the top of the source code module and can be imported by other Python code. Each namedtuple is appended to a list as the sentence is scanned. After scanning completes, the list of `TimeValue` namedtuples is converted to JSON and returned to the caller.

Time Expression Formats

Using notation similar to that used by the PHP time reference, as well as the Wikipedia article on ISO 8601 formats, we define the following quantities:

Short-hand	Meaning
h	hour digit, 0-9
h12	12 hr. clock, hours only, 0-9
h24	24 hr. clock, hours only, zero-padded, 00-24
m	minutes digit, 0-9
mm	minutes, zero-padded, 00-59
ss	seconds, zero-padded 00-60 (60 means leap second)
am_pm	am or pm designator, can be `am` or `pm`, either lower or upper case, with each letter optionally followed by a `.` symbol
t	either `t` or `T`
f	fractional seconds digit
?	optional
utc_time	`hh`, `hh:mm`, `hhmm`, `hh:mm:ss`, `hhmmss`, `hh:mm:ss.ffffff`, `hhmmss.ffffff`

With these definitions, the time expression formats that ClarityNLP recognizes are:

Time Expression Format	Examples
utc-timeZ	10:14:03Z
utc_time+-hh:mm	10:14:03+01:30, 10:14:03-01:30
utc_time+-hhmm	10:14:03+0130, 10:14:03-0130
utc_time+-hh	10:14:03+01, 10:14:03-01
YYYY-MM-DDTHH:MM:SS(.ffffff)?	1969-07-20T10:14:03.123456
(here MM:SS means minutes and seconds)	
h12 am_pm	4 am, 5PM, 10a.m., 9 pm.
h12m am_pm	5:09 am, 9:41 P.M., 10:02 AM.
h12ms am_pm	06:10:37 am, 10:19:36P.M., 1:02:03AM
h12msf	7:11:39:012345 am, 11:41:22.22334 p.m.
h12m	4:08, 10:14, and 11:59
t?h24m	14:12, 01:27, 10:27, T23:43
t?h24ms	01:03:24, T14:15:16
t?h24msf	04:08:37.81412, 19:20:21.532453, 08:11:40:123456
t?hhmm	0613, t0613
t?hhmmss	232120, 120000
t?h24ms with timezone abbreviation	040837CEST, 112345 PST, T093000 Z
t?h24ms with GMT offset	T192021-0700, 14:45:15+03:30

A list of world time zone abbreviations can be found here. ClarityNLP supports this list as well as `Z`, meaning "Zulu" or UTC time.

Finding Size Measurements

Overview

Size measurements are common in electronic health records, especially in radiology and other diagnostic reports. By 'size measurement' we mean a 1D, 2D, or 3D expression involving lengths, such as:

Example	Meaning
3mm	1D measurement
1.2 cm x 3.6 cm	2D measurement
3 by 4 by 5 cm	3D measurement
1.5 cm2	area measurement
4.3 mm3	volume measurement
2.3 - 4.5 cm	range of lengths
1.1, 2.3, 8.5, and 12.6 cm	list of lengths
1.5cm craniocaudal x 2.2cm transverse	measurement with views

ClarityNLP scans sentences for size measurements, extracts the numeric values for each dimension, normalizes each to a common set of units (performing unit conversions if necessary), and provides output in JSON format to other pipeline components.

Source Code

The source code for the size measurement finder module is located in `nlp/algorithms/finder/size_measurement_finder.py`.

Inputs

A single string, the sentence to be scanned for size measurements.

Outputs

A JSON array containing these fields for each size measurement found:

Field Name	Explanation
text	text of the complete measurement
start	offset of the first character in the matching text
end	offset of the final character in the matching text plus 1
temporality	CURRENT or PREVIOUS, indicating when the measurement occurred
units	either mm, mm2, or mm3
condition	either 'RANGE' for numeric ranges, or 'EQUAL' for all others
x	numeric value of first number
y	numeric value of second number
z	numeric value of third number
values	for lists, a JSON array of all values in the list
xView	view specification for the first axis
yView	view specification for the second axis
zView	view specification for the third axis
minValue	either `min([x, y, z])` or `min(values)`
maxValue	either `max([x, y, z])` or `max(values)`

All JSON measurement results contain an identical number of fields. Any fields that are not valid for a given measurement will have a value of EMPTY_FIELD and should be ignored.

All string operations of the size measurement finder are case-insensitive.

Algorithm

ClarityNLP uses a set of regular expressions to recognize size measurements. It scans a sentence with each regex, keeps track of any matches, and finds the longest match among the matching set. The longest matching text string is then tokenized, values are extracted, units are converted, and a python namedtuple representing the measurement is generated. This process is repeated until no more measurements are found, at which point the array of measurement namedtuples is converted to JSON and returned to the caller.

Measurement Formats

ClarityNLP is able to recognize size measurements in a number of different formats. Using notation similar to that of[1], we define the following quantities:

[1]

M. Sevenster, J. Buurman, P. Liu, J.F. Peters, P.J. Chang
Natural Language Processing Techniques for Extracting and Categorizing Finding Measurements in Narrative Radiology Reports
Appl. Clin. Inform., 6(3) 600-610, 2015.

Shorthand	Meaning
x y z	Any numeric value, either floating point or integer
cm	Units for the preceding numeric value
by	Either the word 'by' or the symbol 'x'
to	Either the word 'to' or the symbol '-'
vol	Dimensional modifier, either 'square', 'cubic', 'sq', 'sq.', 'cu', 'cu.', 'cc'
view	View specification, any word will match

With these definitions, the measurement formats that ClarityNLP recognizes are:

Regex Form	Examples
x cm	3 mm, 5cm, 10.2 inches
x vol cm	5 square mm, 3.2cm2
x to y cm	3-5 cm, 3 to 5cm
x cm to y cm	3 cm to 5 cm, 3cm - 5 cm
x by y cm	3 x 5 inches, 3x5 cm
x cm by y cm	3 mm by 5 mm
x cm view by y cm view	3 cm craniocaudal x 5 cm transverse
x by y by z cm	3 x 5 x 7 mm
x by y cm by z cm	3 x 5mm x 7 mm
x cm by y cm by z cm	3 mm x 5 mm x 7 mm
x cm view by y cm view by z cm view	3 cm craniocaudal by 5cm transverse by 7 cm anterior

ClarityNLP can also find size measurements with nonuniform spacing between the various components, as several of the examples above demonstrate. Newlines can also be present within a measurement. Inconsistent spacing such as this appears frequently in electronic health records.

Details

These medically-relevant measurement units are supported:

Units	Textual Forms
millimeters	mm, millimeter, millimeters
centimeters	cm, centimeter, centimeters
inches	in, inch, inches

ClarityNLP tries to distinguish uses of the word 'in' as a preposition vs. its use as a unit of length. **It cannot correctly identify all such instances.** Hence the word 'in' preceded by a numeric value may sometimes generate false positive results.

Numeric values can be integers (sequence of digits) or floating point values. The digit before the decimal point is optional. Some examples:

- 3, 42
- 12.4887
- .314, 0.314

References

Extracting Tumor Stage Information

Overview

The Union for International Cancer Control (UICC) has developed a system for classifying malignant tumors called the TNM staging system. Each tumor is assigned an alphanumeric code (the TNM code) that describes the extent of the tumor, lymph node involvement, whether it has metastasized, and several other descriptive factors. The code also includes staging information. ClarityNLP can locate these codes in medical reports and decode them. This document describes the TNM system and the information that ClarityNLP provides for each TNM code that it recognizes.

Information on the TNM system was taken from the reference document[1] and the explanatory supplement[2]. Information on serum marker values was taken from the Wikipedia article on the TNM staging system[3].

Source Code

The source code for the TNM stage module is located in `nlp/algorithms/value_extraction/tnm_stage_extractor.py`.

Inputs

A single string representing the sentence to be searched for TNM codes.

Outputs

A JSON array containing these fields for each code found:

[1] J. Brierly, M. Gospodarowicz, C. Wittekind, *eds.*
TNM Classification of Malignant Tumors, Eighth Edition
Union for International Cancer Control (UICC)
Wiley Blackwell, 2017
https://www.uicc.org/resources/tnm

[2] C. Wittekind, C. Compton, J. Brierly, L. Sobin, *eds.*
TNM Supplement: A Commentary on Uniform Use
Union for International Cancer Control (UICC)
Wiley Blackwell, 2012

[3] https://en.wikipedia.org/wiki/TNM_staging_system

Field Name	Explanation
text	text of the complete code
start	offset of first char in the matching text
end	offset of final char in the matching text + 1
t_prefix	see prefix code table below
t_code	extent of primary tumor
t_certainty	primary tumor certainty factor
t_suffixes	see T suffix table below
t_multiplicity	tumor multiplicity value
n_prefix	see prefix code table below
n_code	regional lymph node involvement
n_certainty	certainty factor for lymph node involvement
n_suffixes	see N suffix table below
n_regional_nodes_examined	number of regional lymph nodes examined
n_regional_nodes_involved	number of regional lymph nodes involved
m_prefix	see prefix code table below
m_code	distant metastasis
m_certainty	certainty factor for distant metastasis
m_suffixes	see M suffix table below
l_code	lymphatic invasion code
g_code	histopathological grading code
v_code	venous invasion code
pn_code	perineural invasion code
serum_code	serum tumor marker code
r_codes	residual metastases code
r_suffixes	see R suffix table below
r_locations	string array indicating location(s) of metastases
stage_prefix	see prefix table below
stage_number	integer value of numeric stage
stage_letter	supplementary staging information

All JSON measurement results contain an indentical number of fields. Any fields that are not valid for a given measurement will have a value of EMPTY_FIELD and should be ignored.

Algorithm

ClarityNLP uses a set of regular expressions to recognize TNM codes as a whole and to decode the individual subgroups. A TNM code consists of mandatory T, N, and M groups, as well as optional G, L, R, Pn, S, and V groups. A staging designation may also be present.

Prefixes

The set of prefixes used for the groups is found in the next table:

Prefix Letter	Meaning
c	clinical classification
p	pathological classification
yc	clinical classification performed during multimodal therapy
yp	pathological classification performed during multimodal therapy
r	recurrent tumor
rp	recurrence after a disease-free interval, designated at autopsy
a	classification determined at autopsy

Certainty Factor

The T, N, and M groups can have an optional certainty factor, which indicates the degree of confidence in the designation. This certainty factor was present in the 4th through 7th editions of the TNM guide, but it has been removed from the 8th edition[1].

Certainty Factor	Meaning
C1	evidence from standard diagnostic means (inspection, palpitation)
C2	evidence from special diagnostic means (CT, MRI, ultrasound)
C3	evidence from surgical exploration, including biopsy and cytology
C4	evidence from definitive surgery and pathological examination
C5	evidence from autopsy

T Group

The T group describes the extent of the primary tumor:

T Code	Meaning
TX	primary tumor cannot be assessed
T0	no evidence of primary tumor
Tis	carcinoma *in situ*
T1, T2, T3, T4	increasing size and/or local extent of primary tumor

For multiple tumors, the multiplicity appears in parentheses after the T group code, e.g. `T1(m)` or `T1(3)`. Anatomical subsites are denoted with suffixes `a`, `b`, `c`, or `d`, e.g. `T2a`. Recurrence in the area of a primary tumor is denoted with the + suffix.

N Group

The N group describes the extent of regional lymph node involvement:

N Code	Meaning
NX	reginal lymph node involvement cannot be assessed
N0	no regional lymph node metastasis
N1, N2, N3	increasing involvement of regional lymph nodes

Anatomical subsites are denoted with suffixes `a`, `b`, `c`, or `d`, e.g. `N1b`. With only micrometastasis (smaller than 0.2 cm), the suffix `(mi)` should be used, e.g. `pN1(mi)`.

Suffix `(sn)` indicates sentinal lymph node involvement.

Examination for isolated tumor cells (ITC) is indicated with the suffixes in parentheses (e.g. `pN0(i-)`):

ITC Suffix	Meaning
(i-)	no histologic regional node matastasis, negative morphological findings for ITC
(i+)	no histologic regional node metastasis, positive morphological findings for ITC
(mol-)	no histologic regional node metastasis, negative non-morphological findings for ITC
(mol+)	no histologic regional node metastasis, positive non-morphological findings for ITC

Examination for ITC in sentinel lymph nodes uses these suffixes:

ITC(sn) Suffix	Meaning
(i-)(sn)	no histologic sentinel node matastasis, negative morphological findings for ITC
(i+)(sn)	no histologic sentinel node metastasis, positive morphological findings for ITC
(mol-)(sn)	no histologic sentinel node metastasis, negative non-morphological findings for ITC
(mol+)(sn)	no histologic sentinel node metastasis, positive non-morphological findings for ITC

The TNM supplement[2] chapter 1, p. 8 recommends adding the number of involved and examined regional lymph nodes to the `pN` classification (pathological classification), e.g. `pN1b(2/11)`. This example says that 11 regional lymph nodes were examined and two were found to be involved.

M Group

The M group describes the extent of distant metastasis:

M Code	Meaning
MX	metastasis cannot be assessed; considered inappropriate if metastasis can be evaluated based on physical exam alone; see[1] p. 24,[2] pp. 10-11.
M0	no distant metastasis
M1	distant metastasis
pMX	invalid category ([2], p. 10)
pM0	only to be used after autopsy ([2], p. 10)
pM1	distant metastasis microscopically confirmed

The `M1` and `pM1` subcategories may be extended by these optional suffixes, indicating the location of the distant metastasis:

Location Suffix	Meaning
PUL	pulmonary
OSS	osseous
HEP	hepatic
BRA	brain
LYM	lymph nodes
MAR	bone marrow
PLE	pleura
PER	peritoneum
ADR	adrenals
SKI	skin
OTH	other

Anatomical subsites are denoted with suffixes `a`, `b`, `c`, and `d`. The suffix `(cy+)` is valid for `M1` codes under certain conditions (see[2] p. 11).

For isolated tumor cells (ITC) found in bone marrow ([2] p. 11), these suffixes can be used:

Suffix	Meaning
(i+)	positive morphological findings for ITC
(mol+)	positive non-morphological findings for ITC

R Group

The R group describes the extent of residual metastases:

R Code	Meaning
RX	presence of residual tumor cannot be assessed
R0 (location)	residual tumor cannot be detected by any diagnostic means
R1 (location)	microscopic residual tumor at indicated location
R2 (location)	macroscopic residual tumor at indicated location

The TNM supplement ([2], p. 14) recommends annotating R with the location in parentheses, e.g. `R1 (liver)`. There can also be multiple R designations if residual tumors exist in more than one location.

The presence of noninvasive carcinoma at the resection margin should be indicated by the suffix `(is)` (see[2], p. 15).

The suffix `(cy+)` for R1 is valid under certain conditions ([2], p. 16).

G Group

The G group discribes the histopathological grading score and has these values:

G Code	Meaning
GX	grade of differentiation cannot be assessed
G1	well differentiated
G2	moderately differentiated
G3	poorly differentiated
G4	undifferentiated

`G1` and `G2` may be grouped together as `G1-2` ([2], p. 23).
`G3` and `G4` may be grouped together as `G3-4` ([2], p. 23).

L Group

The L group indicates whether lymphatic invasion has occurred:

L Code	Meaning
LX	lymphatic invasion cannot be assessed
L0	no lymphatic invasion
L1	lymphatic invasion

V Group

The V group indicates whether venous invasion has occurred:

V Code	Meaning
VX	venous invasion cannot be assessed
V0	no venous invasion
V1	microscopic venous invasion
V2	macroscopic venous invasion

Pn Group

The Pn group indicates whether perineural invasion has occurred:

Pn Code	Meaning
PnX	perineural invasion cannot be assessed
Pn0	no perinerual invasion
Pn1	perineural invasion

Serum Group

The S group indicates the status of serum tumor markers:

S Code	Meaning
SX	marker studies not available or not performed
S0	marker study levels within normal limits
S1	markers are slightly raised
S2	markers are moderately raised
S3	markers are very high

Staging

The staging value indicates the severity of the tumor. A staging assignment depends on the tumor type and is indicated either with digits or roman numerals, and optionally with subscript `a`, `b`, `c`, or `d`. The stage designation can also have a `y` or `yp` prefix as well ([2], p. 18).

References

General Value Extraction

Overview

Value extraction is the process of scanning text for query terms and finding numeric values associated with those terms. For example, consider the sentence:

```
The patient's heart rate was 60 beats per minute.
```

It is clear that the value 60 is associated with `heart rate`. A value extractor using this sentence as input should therefore return 60 as the result for the query `heart rate`.

Values can occur either before or after the query terms, since both variants are acceptable forms of English expression:

```
A 98.6F temperature was measured during the exam.    (before)
A temperature of 98.6F was measured during the exam. (after)
```

The *value-follows-query* form is dominant in the text of medical records. To constrain the scope of the problem and to reduce the chances of error:

ClarityNLP assumes that the value FOLLOWS the query terms.

This assumption does **not** imply anything about the distance between the query and the value. Sometimes the value immediately follows the term, as in terse lists of vital signs:

```
Vitals: Temp 100.2 HR 72 BP 184/56 RR 16 sats 96% on RA
```

Other times, in narrative text, one or more words fill the space between query term and value:

```
The temperature recorded for the patient at the exam was 98.6F.
```

ClarityNLP tries to understand these situations and correctly associate the value 98.6 with "temperature".

We should emphasize that this is a **generic** value extractor. Our design goal is to achieve good performance across a wide variety of value extraction problems. It has **not** been specialized for any particular type of problem, such as for extracting temperatures or blood pressures. It instead uses an empirically-determined set of rules and regular expressions to find values (either numeric or textual - see below) that are likely to be associated with the query terms. These regexes and rules are under continual refinement and testing as the development of ClarityNLP continues.

You can get a clearer picture of what the value extractor does and the results that it finds by examining our comprehensive suite of value extractor tests.

Value Types

The value extractor can recognize several different value types:

Value Type	Example
Nonnegative Integer	0, 3, 42
Nonnegative Floating Point	3.1415, .27, 0.27
Numeric Range	2-5, 2.3 - 4.6, 2.3 to 4.6
Numeric Range with Matching Units	15 ml to 20 ml
Fraction	120/80, 120 / 80, 120 /80
Fraction Range	110/70 - 120/80

Fractions can have arbitrary whitespace on either side of the forward slash, as some of these examples illustrate. For floating point numbers, the digit before the decimal point is optional.

Value Relationships

The value extractor can associate queries and values expressed in many different formats:

Format	Example
No space	`T98.6`
Whitespace	`T 98.6`, `T 98.6`
Dash	`T-98.6`, `T- 98.6`
Colon	`T:98.6`, `T :98.6`
Equality	`T=98.6`, `T = 98.6`, `T =98.6`, `T is 98.6`
Approximations	`T ~ 98.6`, `T approx. 98.6`, `T is ~98.6`
Greater Than or Less Than	`T > 98.6`, `T<=98.6`, `T .lt. 98.6`, `T gt 98.6`
Narrative	`T was greater than 98.6`

These are just a few of the many different variants that the value extractor supports. In general, the amount of whitespace between query and value is arbitrary.

Result Filters

Numerical results can be filtered by user-specified min and max values. Any results that fall outside of the interval `[min, max]` are discarded. Any numeric value is accepted if these limits are omitted in the NLPQL statement.

For fractions, the value extractor returns the numerator value by default. The denominator can be returned instead by using the `is_denom_only` argument (see below).

Hypotheticals

The value extractor attempts to identify hypothetical phrases and to ignore any values found therein. It uses a simplified version of the *ConText* algorithm of[1] to recognize hypothetical phrases. The "trigger" terms that denote the start of a hypothetical phrase are: `in case`, `call for`, `should`, `will consider`, and `if` when not preceded by `know` and not followed by `negative`.

Source Code

The source code for the value extractor module is located in `nlp/algorithms/value_extraction/value_extractor.py`.

Inputs

The entry point to the value extractor is the `run` function:

```
def run(term_string,                # string, comma-separated list of query terms
        sentence,                   # string, the sentence to be processed
        str_minval=None,            # minimum numeric value
        str_maxval=None,            # maximum numeric value
        str_enumlist=None,          # comma-separated string of terms (see below)
```

(continues on next page)

[1] H. Harkema, J. Dowling, T. Thornblade, W. Chapman
ConText: an Algorithm for Determining Negation, Experiencer, and Temporal Status from Clinical Reports
J. Biomed. Inform., 42(5) 839-851, 2009.

(continued from previous page)

```
        is_case_sensitive=False,   # set to True to preserve case
        is_denom_only=False)       # set to True to return denoms
```

If the `str_minval` and `str_maxval` arguments are omitted, ClarityNLP accepts any numeric value that it finds for a given query. The `str_enumlist` argument will be explained below. The other arguments should be self-explanatory.

Outputs

A JSON array containing these fields for each value found:

Field Name	Explanation
sentence	the sentence from which values were extracted
terms	comma-separated list of query terms
querySuccess	"true" if a value was found, "false" if not
measurementCount	the number of values found
measurements	array of results

Each result in the measurements array contains these fields:

Field Name	Explanation
text	matching text containing query and value
start	offset of the first character in the matching text
end	offset of the final character in the matching text plus 1
condition	a string expressing the relation between query and value: APPROX, LESS_THAN, LESS_THAN_OR_EQUAL, GREATER_THAN, GREATER_THAN_OR_EQUAL, EQUAL, RANGE, FRACTION_RANGE
matchingTerm	the query term associated with this value
x	matching value
y	matching value (only for ranges)
minValue	minimum value of x and y
maxValue	maximum value of x and y

All JSON results will have an identical number of fields. Any fields that are not valid for a given result will have a value of EMPTY_FIELD and should be ignored.

Text Mode and the Enumeration List

The value extractor supports a mode of operation ("text mode") in which it extracts text strings instead of numeric values. Text mode can be enabled by supplying a comma-separated string of terms to the *enum_list* parameter in your NLPQL statement. The enumlist acts like a term filter for the results. Only those terms appearing in the enumlist are returned in the `value` field of the JSON result.

To illustrate how text mode works, suppose you have the task of searching medical records for the presence of hepatitis B or C infections. You want to use ClarityNLP to scan the data and report any lab results that mention HBV or HCV.

The presence or absence of HBV or HCV is typically reported as either "positive" or "negative", or sometimes as just "+" or "-".

You would start by constructing an enumlist with the terms and symbols that you want, such as `"positive, negative, +, -"`. This string would be supplied as the value for the NLPQL enum_list. Your *termset* would include the strings `"HBV"` and `"HCV"`.

Next suppose that, during a run, ClarityNLP were to encounter the sentence `She was HCV negative, HBV +, IgM Titer-1:80, IgG positive`. The value extractor would process this sentence, noticing the presence of the enumlist, and therefore put itself into text mode. When processing completes the value extractor would return two results. The first JSON result would have these values for the matching "term" and "value" fields (other fields omitted):

```
{
    "term":"HCV",
    "value":"negative"
}
```

The second JSON result would have these values:

```
{
    "term":"HBV",
    "value":"+"
}
```

In this manner the value extractor supports the extraction of textual "values" in addition to numeric values.

Algorithm

The value extractor does its work in four stages. The first stage consists of preprocessing operations; the second stage extracts candidate values; the third stage performs overlap resolution to choose a winner from among the candidates; and the fourth stage removes hypotheticals. All results that remain are converted to JSON format and returned to the caller.

Preprocessing

In the preprocessing stage, a few nonessential characters (such as parentheses and brackets) are removed from the sentence. Removal of these characters helps to simplify the regular expressions at the core of the value extractor. Conversion to lowercase follows for the default case-insensitive mode of operation. Identical preprocessing operations are applied to the list of query terms.

The sentence is then scanned for *date expressions*, *size measurements*, and *time expressions*. The value extractor erases any that it finds, subject to these restrictions:

1. Date expressions are not erased if they consist entirely of simple digits. For instance, the date finder will identify the string "1995" as the year 1995, but "1995" could potentially be a volume measurement or another value in a different context.
2. All size measurements are erased unless the units are cubic centimeters or inches. Measurements in inches are kept since "in" as an abbreviation for "inches" can be easily confused with "in" as a preposition. ClarityNLP makes an attempt at disambiguation, but at present it does not have a technique that works reliably in all instances. Part of speech tagging is generally not helpful either. Tagging algorithms trained on formal Engish text (such as journalism or Wikipedia articles) exhibit lackluster performance on medical text, in our experience.
3. Time measurements require additional processing. Any time measurements that consist entirely of integers on both sides of a − sign are not erased, since these are likely to be numeric ranges instead of time expressions.

ISO time formats such as `hh, hhmm, hhmmss` that are *not* preceded by `at` or `@` are not erased, since these are likely to be values and not time expressions.

Time *durations* such as `2 hrs` are identified and erased.

To illustrate the erasure process, consider this somewhat contrived example:

```
Her BP at 3:27 on3/27 from her12 cm. x9cm x6  cm. heart was110/70.
```

Here we see a sentence containing the time expression `3:27`, a date expression `3/27`, and a size measurement `12 cm. x9cm x6 cm.`. The sentence exhibits **irregular spacing**, as is often the case with clinical text.

Suppose that the query term is `BP`, meaning "blood pressure". When the value extractor processes this sentence, it converts the sentence to lowercase, then scans for dates, measurements, and times. The date and time expressions satisfy the criteria for erasure specified above. The resulting sentence after preprocessing is:

```
her bp at      on     from her                    heart was110/70.
```

This is the text that the value extractor uses for subsequent stages. Observe that the erasure process preserves character offsets.

Candidate Selection

After preprocessing, the value extractor constructs a regular expression for a query involving each search term. **Simple term matching is not sufficient**. To understand why, consider a temperature query involving the term `t`. Term matching would result in a match for every letter t in the text.

The query regex enforces the constraint that the search term can only be found at a word boundary and not as a substring of another word. The query regex accomodates variable amounts of whitespace, separators, and fill words.

The query regex is incorporated into a list of additional regular expressions. These regexes each scan the sentence and attempt to recognize various contexts from which to extract values. These contexts are, with examples:

1. A range involving two fractions connected by "between/and" or "from/to":

```
BP varied from 110/70 to 120/80.
```

2. A range involving two fractions:

```
BP range: 105/75 - 120/70
```

3. A fraction:

```
BP lt. or eq 112/70
```

4. A range with explicit unit specifiers:

```
Platelets between 25k and 38k
```

5. A numeric range involving "between/and" or "from/to":

```
Respiration rate between 22 and 32
```

6. A numeric range:

```
Respiration rate 22-32
```

7. A query of the general form <query_term> <operator> <value>:

```
The patient's pulse was frequently >= 60 bpm.
```

8. A query of the general form <query_term> <words> <value>:

```
Overall LVEF is severely depressed (20%).
```

Multiple regexes typically match a given query, so an overlap resolution process is required to select the final result.

Overlap Resolution

If the value extractor finds more than one candidate for a given query, the overlap resolution process prunes the candidates and selects a winner. The rules for pruning candidates have been developed through many rounds of iterated testing. More rules may be discovered in the future. The situations requiring pruning and the rules for doing so are as follows:

1. **If two candidate results overlap exactly, return the result with the longest matching term.**

   Example:

   sentence:`T=98 BP= 122/58 HR= 7 RR= 20 O2 sat= 100% 2L NC`
   termset:`O2, O2 sat`

   Candiates:

   ```
   {"term":"O2", "value":100, "text":"O2 sat= 100"}
   {"term":"O2 sat", "value":100, "text":"O2 sat= 100"}
   ```

   In this example, both "O2" and "O2 sat" match the value 100, and both matches have identical start/end values. The value extractor returns the candidate for "O2 sat" as the winner since it is the longer of the two query terms and completely encompasses the other candidate.

2. **If two results partially overlap, discard the first match if the extracted value is contained within the search term for the second.**

   Example:

   sentence:`BP 120/80 HR 60-80s RR SaO2 96% 6L NC.`
   termset:`RR, SaO2`

   Candidates:

   ```
   {"term":"RR", "value":2, "text":"RR SaO2 96"}
   {"term":"SaO2", "value":96, "text":"SaO2 96"}
   ```

   Note that the search term `RR` has no matching value in the sentence, so the value extractor keeps scanning and finds the 2 in "SaO2". The 2 is part of a search term, not an independent value, so that candidate result is discarded.

3. (text mode only) **Whenever two results overlap and one result is a terminating substring of the other, discard the candidate with the contained substring.**

   Example:

   sentence:`no enteric gram negative rods found`
   termset:`gram negative, negative`
   enumlist:`rods`

   Candidates:

```
{"term":"gram negative", "value":"rods", "text":"gram negative
rods"}
{"term":"negative", "value":"rods", "text":"negative rods"}
```

The second candidate is a terminating substring of the first and is discarded. Note that this is a different situation from no. 1 above, since the matching text for the candidates have different starting offsets.

4. **If two candidates have overlapping matching terms, keep the candidate with the longest matching term.**

   Example:

   sentence:`BLOOD PT-10.8 PTT-32.6 INR(PT)-1.0`
   termset:`pt, ptt, inr(pt)`

   Candidates:

   ```
   {"term":"pt", "value":10.8, "text":"PT-10.8"}
   {"term":"pt", "value":1.0, "text":"PT)-1.0"}
   {"term":"ptt", "value":32.6, "text":"PTT-32.6"}
   {"term":INR(PT)", "value":1.0, "text":"INR(PT)-1.0"}
   ```

   The second and fourth candidates have overlapping matching query terms. The longest matching term is `INR(PT)`, so candidate four is retained and candidate two is discarded. This is a different situation from no. 3 above, which only applies in text mode.

5. (text mode only) **Keep both candidates if their matching terms are connected by "and" or "or".**

   Example:

   sentence:`which grew gram positive and negative rods`
   termset:`gram positive, negative`
   enumlist:`rods`

   Candidates:

   ```
   {"term":"gram positive", "value":"rods", "text":"gram positive
   and negative rods"}
   {"term":"negative", "value":"rods", "text":"negative rods"}
   ```

   The matching texts for each candidate consts of query terms connected by the word "and", so both results are kept.

6. **If two candidates have overlapping matching text but nonoverlapping query terms, keep the candidate with query term closest to the value.**

   Example:

   sentence:`received one bag of platelets dure to platelet count of 71k`
   termset:`platelets, platelet, platelet count`

   Candidates:

   ```
   {"term":"platelets", "value":71000, "text":"platelets due to
   platelet count of 71k"}
   {"term":"platelet count", "value":71000, "text":"platelet count
   of 71k"}
   ```

   These candidates have overlapping matching texts with nonoverlapping query terms. Keep the candidate with query term "platelet count" since it is closest to the value of 71000.

After these pruning operations, any remaining candidates that express hypothetical conditions (see above) are discarded. The survivor(s) are converted to JSON and returned as the result(s).

In general, users can expect the value extractor to return the first valid numeric result following a query term.

References

Measurement-Subject Resolution

Overview

Measurement-subject resolution is the process of associating size measurements in a sentence with the object(s) possessing those measurements. For instance, in the sentence

```
The spleen measures 7.5 cm.
```

the measurement `7.5 cm` is associated with `spleen`. The word `spleen` is said to be the *subject* of the measurement `7.5 cm`. In this example the subject of the measurement also happens to be the subject of the sentence. This is not always the case, as the next sentence illustrates:

```
The liver is normal in architecture and echogenicity, and is seen to contain
numerous small cysts ranging in size from a few millimeters to approximately
1.2 cm in diameter.
```

Here the subject of the sentence is `liver`, but the subject of the `1.2 cm` measurement is `cysts`.

In this document we describe how ClarityNLP analyzes sentences and attempts to resolve subjects and measurements.

Source Code

The source code for the measurement subject finder is located in `nlp/algorithms/finder/subject_finder.py`.

Inputs

The entry point to the subject finder is the `run` function:

```
def run(term_string,         # string, comma-separated list of query terms
        sentence,            # string, the sentence to be processed
        nosub=False,         # set to True to disable ngram substitutions
        use_displacy=False) # set to True to display a dependency parse
```

The `term_string` argument is a comma-separated list of query terms. The `nosub` argument can be used to disable ngram substitution, described below. The `use_displacy` argument generates an html page displaying a dependency parse of the sentence. This visualization capability should only be used for debugging and development.

Outputs

A JSON array containing these fields for each size measurement found:

Field Name	Explanation
sentence	the sentence from which size measurements were extracted
terms	comma-separated list of query terms
querySuccess	"true" if at least one query term matched a measurement subject
measurementCount	the number of size measurements found
measurements	array of individual size measurements

Each result in the measurements array contains these fields:

Field Name	Explanation
text	text of the complete size measurement
start	offset of the first character in the matching text
end	offset of the final character in the matching text plus 1
temporality	indicartion of when measurement occurred values are 'CURRENT' and 'PREVIOUS'
units	units of the x, y, and z fields values are 'MILLIMETERS', 'SQUARE_MILLIMETERS', and 'CUBIC_MILLIMETERS'
condition	numeric ranges will have this field set to 'RANGE' all other measurements will set this field to 'EQUAL'
matchingTerm	an array of all matching query terms for this measurement
subject	an array of strings, the possible measurement subjects
location	a string representing the anatomic location of the object
x	numeric value of first measurement dimension
y	numeric value of second measurement dimension
z	numeric value of third measurement dimension
values	JSON array of all numeric values in a size list
xView	view specification for x value
yView	view specification for y value
zView	view specification for z value
minValue	minimum value of x, y, and z
maxValue	maximum value of x, y, and z

All JSON results will have an identical number of fields. Any fields that are not valid for a given measurement will have a value of EMPTY_FIELD and should be ignored.

Dependencies

The measurement subject finder has a dependency on ClarityNLP's size measurement finder module, whose documentation can be found here: *Finding Size Measurements*.

There is also a dependency on spaCy, a python library for natural language processing. The spaCy library was chosen for this project because it is fast and produces consistently good results. We will have much more to say about spaCy below.

NGram Generator

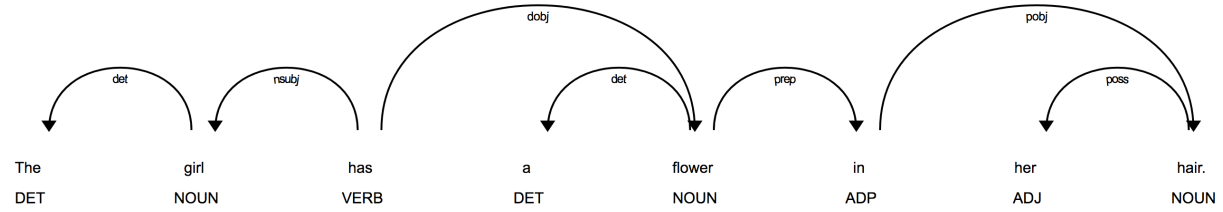
The subject finder module has the option of peforming ngram substitutions with medical ngrams taken from a list (`clarity_ngrams.txt`) that accompanies the source code. This file contains ngrams spanning lengths from 1 to 14 words. The ngrams are stored by length in the file and sorted in decreasing order of length.

The code that generates this file is found in `ngram_gen.py`, also in the same folder. The ngram generator code ingests two source lists of medical terms found in the files `anatomic_sites.txt` and `medra_terms.txt`. These files are parsed, some cleanup is performed, and the lists are sorted and written out as ngrams to `clarity_ngrams.txt`.

The ngrams in `clarity_ngrams.txt` are medical terms that are relatively uncommon in standard English text, such as the text corpora that spaCy's English models were trained on. By replacing uncommon domain-specific terms with more common nouns from everyday English discourse, we have found that we can get substantial improvement in spaCy's ability to analyze medical texts. Several examples below illustrate this substitution process.

The spaCy Dependency Parse

The ClarityNLP subject finder module uses spaCy to generate a *dependency parse* of each input sentence. A dependency parse provides part of speech tags for each word as well as dependency information encoded in tree form. To illustrate, here is a diagram of a dependency parse of the sentence `The girl has a flower in her hair.`



This diagram was generated with spaCy's display tool displacy. The part of speech tags appear underneath each word. In addition to NOUN, VERB, and ADJ, we also see DET (determiner) and ADP (preposition). Documentation on spaCy's annotation scheme can be found here.

The arrows represent a child-parent relationship, with the child being at the "arrow" or "head" end and the the parent at the tail end. The word at the arrow end modifies the word at the tail end. Thus the word `The` modifies `girl`, since the first arrow starts at the word `girl` and points to the word `The`. The label on the arrow indicates the nature of the parent-child relationship. For the "girl-The" arrow, the `det` label on the arrow indicates that the word `The` is a determiner that modifies `girl`.

The subject of the verb `has` is the word `girl`, as indicated by the `nsubj` (nominal subject) label on the second arrow. The direct object of the verb is the noun `flower`, as the arrow labeled `dobj` shows. The direct object has a DET modifer `a`, similarly to the DET modifier for the word `girl`.

A prepositional phrase `in her hair` follows the direct object, as the two arrows labeled `prep` (prepositional modifier) and `pobj` (object of preposition) indicate. The object of the preposition `in` is the noun `hair`, which has a possession modifier `her`.

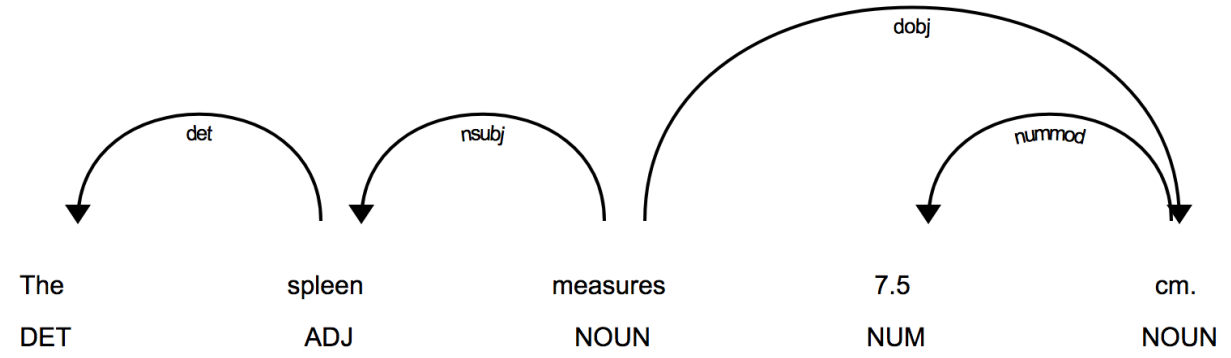
Thus a dependency parse allows one to determine the nature of the relationships between the various components of a sentence. ClarityNLP uses the dependency parse information, along with a set of custom rules and heuristics, to determine the subjects of each size measurement.

Dependency Parse Errors

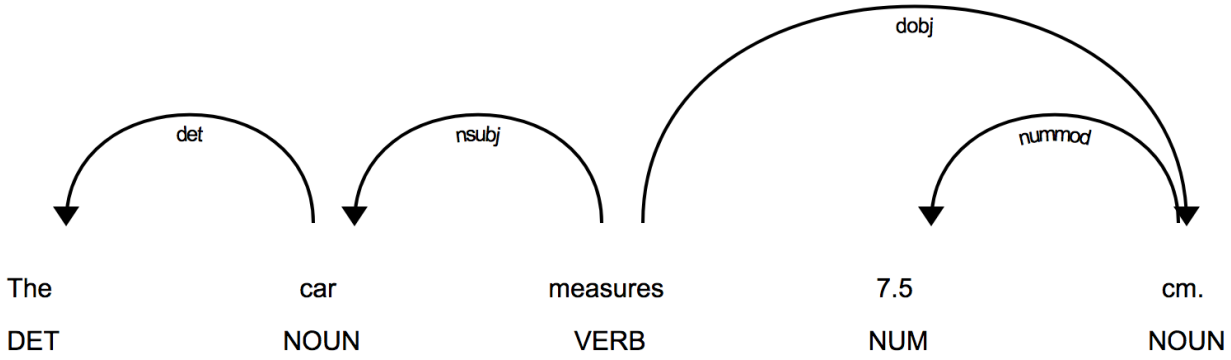
Sometimes spaCy generates an incorrect dependency parse. This happens often in sentences that contain medical terminology, especially when medical terms are used in different contexts from those of the training corpora. For instance, the simple sentence

```
The spleen measures 7.5 cm.
```

has this dependency parse:



Here we see that the verb `measures` was tagged as a noun, in the sense of "weights and measures". The word `spleen` was also tagged as an adjective. This is obviously incorrect. The problem, though, lies with the word `spleen` instead of `measures`. Observe what happens to the dependency parse if `spleen` is replaced by the common noun `car`:



This is the correct result: `car` is tagged as a noun, `measures` is tagged a verb, and the nominal subject of the sentence is `car`.

One can imagine the extent to which obscure medical jargon could completely confuse spaCy. In the absence of a version of spaCy trained on medical texts, ClarityNLP attempts to overcome such problems by replacing medical ngrams with common English nouns. The resulting sentence **does not** have to "make sense". All it needs to do is help spaCy produce the correct dependency parse of the sentence and correctly resolve the relationships between the various phrases. The substitution process is not foolproof either, but we observe consistently better results on medical texts with the ngram substitutions than without them.

To further help spaCy's decision processes, spaCy provides a mechanism for introducing special case tokenization rules. ClarityNLP takes advantage of this by introducing four special case rules for `measure` and related verbs. The next code block shows how ClarityNLP accomplishes this:

```
# 'measures' is a 3rd person singular present verb
special_case = [{ORTH: u'measures', LEMMA: u'measure', TAG: u'VBZ', POS: u'VERB'}]
nlp.tokenizer.add_special_case(u'measures', special_case)

# 'measure' is a non 3rd person singular present verb
special_case = [{ORTH: u'measure', LEMMA: u'measure', TAG: u'VBP', POS: u'VERB'}]
nlp.tokenizer.add_special_case(u'measure', special_case)

# 'measured' is a verb, past participle
special_case = [{ORTH: u'measured', LEMMA: u'measure', TAG: u'VBN', POS: u'VERB'}]
nlp.tokenizer.add_special_case(u'measured', special_case)

```

(continues on next page)

(continued from previous page)

```
# 'measuring' is a verb form, either a gerund or present participle
special_case = [{ORTH: u'measuring', LEMMA: u'measure', TAG: u'VBG', POS: u'VERB'}]
nlp.tokenizer.add_special_case(u'measuring', special_case)
```

Here `ORTH` refers to orthography, the actual sequence of letters appearing in the text. `LEMMA` is the canonical or "dictionary" form of the verb, identical in all cases. The `TAG` entry refers to the part of speech tag using Penn Treebank Notation. The `POS` entry is spaCy's notation for the same part of speech tag.

These rules guarantee that spaCy will interpret the words `measures`, `measure`, `measured`, and `measuring` as verbs.

The words that ClarityNLP substitutes for medical ngrams are:

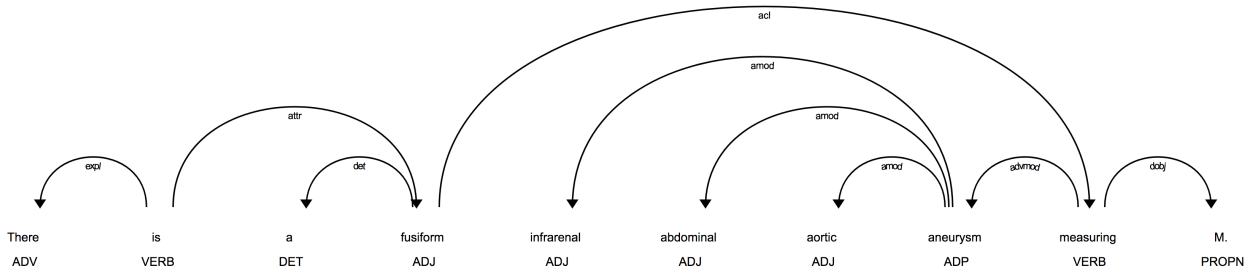
```
car, city, year, news, math, hall, poet, fact,
idea, oven, poem, dirt, tale, world, hotel
```

These are all common English words that only occur as nouns.

One additional illustration can help to make this process clearer. Consider this sentence:

```
There is a fusiform infrarenal abdominal aortic aneurysm measuring M.
```

The dependency parse for this sentence, using the special tokenization rules, is:

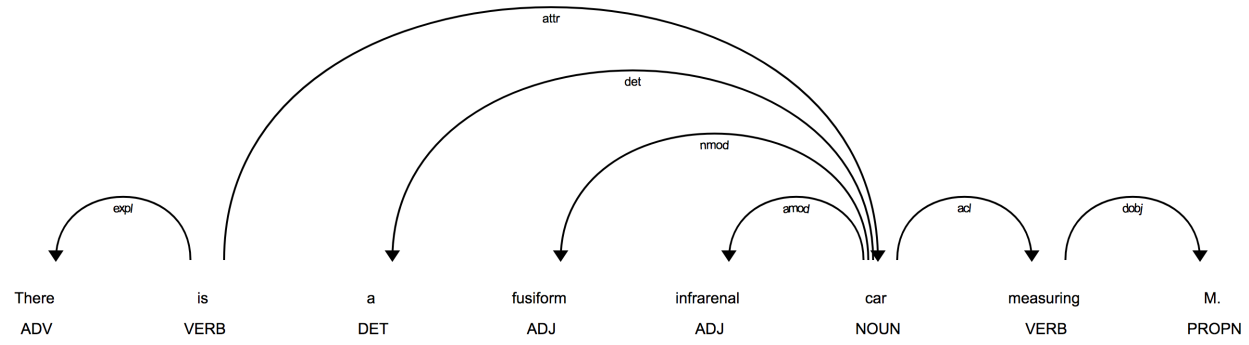


The most obvious problem here is that the word `aneurysm`, which is a noun, has been tagged with `ADP`, indicating either a conjunction or preposition. The adjective `fusiform` was also not deduced to be a modifier of `aneurysm`.

Since the ngram `abdominal aortic aneurysm` is in the ClarityNLP ngram list, substituting `car` for `abdominal aortic aneurysm` produces this sentence:

```
There is a fusiform infrarenal car measuring M.
```

The dependency parse for this new sentence is:



Here we see that the word `car`, a very common English word, has been correctly tagged as a noun. The adjective `fusiform` now modifies car, as it should. The ngram substitution has thus helped spaCy produce a correct de-

pendency parse. Even though the sentence doesn't make sense, the purpose of the substitutions is not to preserve the meaning of the sentence. Substitutions help spaCy generate a **correct dependency parse**, allowing the proper relationships among the various sentence components to be determined.

Algorithm

ClarityNLP uses several stages of processing in its attempt to resolve the subject of each size measurement. These processing stages are:

- Sentence cleanup and ngram substitution
- Sentence template determination
- Dependency parse analysis and selection of candidate subjects
- Subject resolution and location determination
- Ngram replacement and JSON conversion

Sentence Cleanup and NGram Substitution

The cleanup stage attempts to simplify the sentence as much as possible. A shorter sentence is more likely to be parsed correctly than a needlessly verbose sentence. Thus ClarityNLP removes all extraneous text from the sentence that has no bearing on the measurement-subject resolution problem. Thse removals include:

- Removing image annotations, such as `(image 302:33), (782b:49)`
- Removing anything in square or curly brackets, such as anonymized dates
- Removing excess verbosity, such as "for example", "in addition", "no evidence of", etc.
- Replacing verbose forms with less verbose forms, such as:
  - "measuring upwards of" => "measuring"
  - "is seen to contain" => "contains"
  - "is seen in" => "in"
  - etc.
- Replacing roman numerals with decimal numbers
- Replacing semicolons with whitespace (misplaced semicolons can have a deleterious effect on the dependency parse)
- Substituting simple nouns for medical ngrams
- Collapsing repeated whitespace into a single space
- Finding size measurements and replacing the measurement text with `M`

This last item deserves some explanation. The sentence

```
The spleen measures 7.5 cm.
```

is transformed by the measurement replacement operation to this:

```
The spleen measures M.
```

The reason for the M-replacement is to facilitate the recognition of sentence patterns in the text. We call these sentence patterns "sentence templates". Sentences that fit a common template pattern provide clues about the sentence structure and can be analyzed in identical ways. For instance, size measurements in medical texts are often reported as

```
{Something} measures {size_measurement}.
```

Some examples:

```
The spleen is unremarkable measuring 8.6 cm.
The cyst in the upper pole of the kidney measures 1.2 cm.
The duct tapers smoothly to the head of the pancreas,
where it measures approximately 5 mm.
```

After M-replacement, these sentences become:

```
The spleen is unremarkable measuring M.
The cyst in the upper pole of the kidney measures M.
The duct tapers smoothly to the head of the pancreas,
where it measures approximately M.
```

A regular expression designed to find a capital M preceded by a measurement verb could easily identify all of these sentences as belonging to the same underlying template. Custom rules for each matching sentence could be applied to resolve the object having measurement M. ClarityNLP uses this approach for this template and the others described below.

Sentence Template Determination

ClarityNLP uses a set of sentence patterns or templates to help it resolve measurements and their subjects. These templates were determined by examining a large number of electronic health records and noting common forms of expression. A set of regular expressions was developed for classifying sentences into the various patterns. This set of regexes and sentence patterns will likely expand as ClarityNLP evolves.

For the discussion below, it is helpful to define a few items, using a notation similar to that for regular expressions:

Abbreviation	Examples
MEAS	"measure", "measures", "measured", "measuring"
WORD	a word or number, with optional punctuation and spaces
||	string concatenation
*	zero or more of the previous item
+	one or more of the previous item
*?, +?	nongreedy version of * or +
M	size measurement
Q	measurement qualifiers: "all", "approximately", "currently", "mainly", etc.
DET	determiners: "a", "an", "the"
TERMINATOR	"a", "an", or MEAS

The templates used by ClarityNLP are:

1. Subject Measures M

This template, illustrated above, recognizes sentences or sentence fragments containing an explicit measurement verb. The subject of the measurement M is generally in the set of words preceding MEAS.

Pattern:

WORD+ || MEAS || WORD* || M

2. DET Words M

This template recognizes sentences or sentence fragments that omit an explicit measurement verb. For instance:

"An unchanged 2cm hyperechoic focus..."
"...and has a simple 1.2 x 2.9 x 2.9 cm cyst..."

Greedy and nongreedy patterns:

DET || WORD+ || Q* || M || WORD+
DET || WORD+ || Q* || M || WORD+? || TERMINATOR

3. DET M Words

Same as #2, but with the words in a different order. Examples:

"A 3cm node in the right low paratracheal station..."
"The approximately 1 cm cyst in the upper pole of the left kidney..."

Greedy and nongreedy patterns:

DET || Q* || M || WORD+
DET || Q* || M || WORD+? || TERMINATOR

4. Ranging in Size

The phrase "ranging in size" occurs frequently in diagnostic medical reports. ClarityNLP substitutes the verb "measuring" for "ranging in size" and then applies the *Subject Measures M* template to the sentence. An example:

"Distended gallbladder with multiple stones ranging in size from a few millimeters to 1 cm in diameter."

5. Now vs. Then

This template recognizes sentences comparing measurements taken on different dates. For instance:

"The lesion currently measures 1.3 cm and previously measured 1.2 cm."
"A left adrenal nodule measures 1.2 cm as compared to 1.0 cm previously."

ClarityNLP uses a set of seven regexes in its attempts to find such sentences. The first regex is used to match the first measurement of the pair, and the others are used to match the second measurement.

6. Before and After

This template recognizes sentences and sentence fragments with measurement subjects occurring before and after each measurement. For example:

"The left kidney measures 8.5 cm and contains an 8 mm x 8 mm anechoic rounded focus along the lateral edge, which is most likely a simple renal cyst."

Pattern:

DET || WORDS+ | MEAS || Q* || M || WORD* || DET || M || WORDS+

ClarityNLP searches for measurement subjects in each WORDS+ group captured by the associated regex.

7. M and M

This template recognizes sentences comparing two similar objects, two views of an object, or an object and features inside it. For instance:

"The lower trachea measures 14 x 8 mm on expiratory imaging and 16 x 17 mm on inspiratory imaging."

"The largest porta hepatis lymph node measures 1.6 cm in short axis and 2.6 cm in long axis."

Pattern 1:

WORD* || MEAS || Q* || M || WORD* || and || WORD*

Pattern 2:

WORD+ || MEAS || Q* || M || WORD* || and || WORD+ || to || Q* || M || WORD+

8. Carina

This is a special case template for sentences involving endotracheal tubes and distances relative to the carina. An example sentence:

"Endotracheal tube is in standard position about 5 cm above the carina."

Template Matching

ClarityNLP counts the number of M's in the sentence after the cleanup phase and attempts template matching on fragments containing either one or two M's. Sentences or fragments matching a template are sent to the next stage of processing, dependency parse analysis, described below. If no templates match, ClarityNLP attempts a dependency parse analysis without having the benefit of knowing the sentence structure via a template match. ClarityNLP will attempt measurement-subject resolution on sentences containing as many as three measurements.

Dependency Parse Analysis

After the template matching phase completes, ClarityNLP uses spaCy to generate a dependency parse of the sentence or fragment that matched the template. ClarityNLP uses the dependency parse information and a set of custom rules to navigate the parse tree looking for the measurement subject. This is typically the noun modified by the measurement itself. For simple sentences this noun is relatively easy to find, since it is often the subject of the sentence. For more

complex sentences, ClarityNLP must navigate the (sometimes incorrect) parse tree using a set of heuristics, custom rules, and corrective actions in an attempt to find the subject. The actual algorithm itself is complex and involves handling of many specal cases, many of which were developed to correct errors in the parse tree. The full algorithm can be found in the function `get_meas_subject` in the file `nlp/finder/subject_finder.py`.

Finding the Starting Token

ClarityNLP begins its examination of the parse tree by searching for the token with text "M" (which has replaced the measurement(s)). If this token is not its own parent, meaning that it is a child node of another token, Clarity starts its processing with the parent of the M node. If the M node *is* its own parent, ClarityNLP looks for the verb token nearest the M token as its starting point. If a verb cannot be found, ClarityNLP looks for a dependency of `nsubj` or `compound` and takes whichever it can find. If none of these can be found, ClarityNLP gives up on finding a starting token and returns an empty subject.

Navigating the Parse Tree

After finding a starting token, ClarityNLP then begins to navigate the parse tree, searching for a measurement subject. Both the part of speech tag and the dependency relationship contribute to ClarityNLP's decision at each node.

The first determination ClarityNLP makes is whether it has arrived at the root node or not. If it happens to be at the root node, it can go no further in the tree, so it looks for a measurement subject (noun) amongst the children of the root node, if any.

If a verb is encountered when navigating the parse tree, a check is made on the dependency for the verb token. If it is "nsubj", meaning the nominal subject of the sentence, experimentation suggests that the part of speech tag was probably incorrectly set to VERB instead of NOUN. The token is saved and used as a candidate subject. If the verb is a measurement verb, the parent token is selected as a candidate subject.

If a noun is encountered, ClarityNLP's decision depends on the dependency label for the token. Some dependency relationships are ignorable, which means that the parent node linked to a child with an ignorable dependency cannot be the measurement subject. These ignorable dependency relationships are:

Dependency	Meaning
acomp	adjectival complement
attr	attribute
conj	conjunct
dobj	direct object
pcomp	complement of preposition
pobj	object of preposition
prep	preposition

Any noun token linked to its parent via an ignorable dependency is skipped, and ClarityNLP moves up one level in the tree to the parent node.

ClarityNLP applies several other empirically determined rules for handling special cases, such as when it encounters the preposition "with". Normally prepositions are ignored during tree navigation by continuing on to their parent node. The word "with" deserves special handling, because sometimes it is used as a conjunction to link two clauses that could have been independent sentences. To illustrate, consider these sentences:

"There is extensive, pronounced cervical lymphadenopathy throughout levels II through IV, **with** lymph nodes measuring up to 2 cm."

"... as well as a more confluent plaque-like mass **with** a broad base along the tentorial surface measuring approximately 2 cm in greatest dimension."

In the first example, the preposition "with" separates two independent clauses and is used as a conjuction. The subject of the 2 cm measurement is "lymph nodes", which happens to be the object of the preposition "with". In this case the objects of the preposition "with" cannot be ignored.

In the second example, the preposition "with" has an object that can be ignored. The subject of the 2 cm measurement, "mass", is not part of the prepositional phrase associated with the word "with".

ClarityNLP is not always able to resolve these two usages of "with" in all instances. So whenever it encounters the preposition "with", it saves the object of that preposition as a candidate measurement subject and continues navigating the tree.

Subject Resolution and Location Determination

The preceding phase of processing results in a list of candidate subjects. If the list is empty, ClarityNLP was unable to find a subject. If the list is nonempty, any duplicates are removed. If only one subject remains it is chosen as the subject.

If multiple candidate subjects remain, the noun chunks obtained from spaCy's analysis of the sentence helps to select the best candidate. The chunks containing each candidate subject are found, and the distance (in words) from the measurement verb (if any) and the associated measurement are computed. ClarityNLP then chooses the candidate that is either within the same noun chunk as the measurement, or which is the closest candidate to that particular chunk.

ClarityNLP also attempts to find the anatomical location for each measurement subject. To do so, it uses information from the template match to identify the most likely sentence fragment that could contain the location. A set of location-finding regexes then attempts to match the fragment and identify the location. Various special-case rules are applied to any matches found, to remove any matches that happen to not actually be locations, and to remove extraneous words. Any remaining text then becomes the location for the measurement.

If location matching fails for all sentence fragments, or if the sentence failed to match a template altogether, ClarityNLP makes one final attempt to determine a location on the sentence as a whole, using the location-finding regexes and the process described above.

Ngram replacement and JSON conversion

The final stage of processing adds additional modifiers to the chosen subject. ClarityNLP performs a recursive depth-first search through the parse tree to capture all modifiers of the subject, any modifiers of the modifiers, etc. A depth-first search is needed to keep the modifiers in the proper word order as they are discovered.

After all modifiers of the subject have been found, the ngram substitution process is reversed, restoring the original words of the sentence. The list of measurements, along with their subjects and locations, is converted to JSON and returned as the result.

NLPQL Expression Evaluation Algorithms

NLPQL Expression Evaluation

Overview

In this section we describe the mechanisms that ClarityNLP uses to evaluate NLPQL expressions. NLPQL expressions are found in `define` statments such as:

```
define hasFever:
    where Temperature.value >= 100.4;

define hasSymptoms:
    where hasFever AND (hasDyspnea OR hasTachycardia);
```

The expressions in each statement consist of everything between the `where` keyword and the semicolon:

```
Temperature.value >= 100.4

hasFever AND (hasDyspnea OR hasTachycardia)
```

NLPQL expressions can either be mathematical or logical in nature, as these examples illustrate.

Recall that the processing stages for a ClarityNLP job proceed roughly as follows:

1. Parse the NLPQL file and determine which NLP tasks to run.
2. Formulate a Solr query to find relevant source documents, partition the source documents into batches, and assign batches to computational tasks.
3. Run the tasks in parallel and write individual task results to MongoDB. Each individual result from an NLP task comprises a *task result document* in the Mongo database. The term *document* is used here in the MongoDB sense, meaning an object containing key-value pairs. The MongoDB 'documents' should not be confused with the Solr source documents, which are electronic health records.
4. Evaluate NLPQL expressions using the **task result documents** as the source data. Write expression evaluation results to MongoDB as separate result documents.

Thus ClarityNLP evaluates expressions **after** all tasks have finished running and have written their individual results to MongoDB. The expression evaluator consumes the task results inside MongoDB and uses them to generate new results from the expression statements.

We now turn our attention to a description of how the expression evaluator works.

The expression evaluator is built upon the MongoDB aggregation framework. Why use MongoDB aggregation to evaluate NLPQL expressions? The basic reason is that ClarityNLP writes results from each run to a MongoDB collection, and it is more efficient to evaluate expressions using MongoDB facilities than to use something else. Use of a non-Mongo evaluator would require ClarityNLP to:

- Run a set of queries to extract the data from MongoDB
- Transmit the query results across a network (if the Mongo instance is hosted remotely)
- Ingest the query results into another evaluation engine
- Evaluate the NLPQL expressions and generate results
- Transmit the results back to the Mongo host (if the Mongo instance is hosted remotely)
- Insert the results into MongoDB.

Evaluation via the MongoDB aggregation framework is more efficient than this process, since all data resides inside MongoDB.

NLPQL Expression Types

In the descriptions below we refer to NLPQL **variables**, which have the form `nlpql_feature.field_name`. The NLPQL feature is a label introduced in a `define` statement. The `field_name` is the name of an output field generated by the task associated with the NLPQL feature.

The output field names from ClarityNLP tasks can be found in the *NLPQL Reference*.

1. Simple Mathematical Expressions

A simple mathematical expression is a string containing NLPQL variables, operators, parentheses, or numeric literals. Some examples:

```
Temperature.value >= 100.4
(Meas.dimension_X > 5) AND (Meas.dimension_X < 20)
(0 == Temperature.value % 20) OR (1 == Temperature.value % 20)
```

The variables in a simple mathematical expression all refer to a **single** NLPQL feature.

Simple mathematical expressions produce a result from data contained in a **single** task result document. The result of the expression evaluation is written to a new MongoDB result document.

2. Simple Logic Expressions

A simple logic expression is a string containing NLPQL features, parentheses, and the logic operators `AND`, `OR`, and `NOT`. For instance:

```
hasRigors OR hasDyspnea
hasFever AND (hasDyspnea OR hasTachycardia)
(hasShock OR hasDyspnea) AND (hasTachycardia OR hasNausea)
(hasFever AND hasNausea) NOT (hasRigors OR hasDyspnea)
```

Logic expressions operate on high-level NLPQL features, **not** on numeric literals or NLPQL variables. The presence of a numeric literal or NLPQL variable indicates that the expression is either a mathematical expression or possibly invalid.

Simple logic expressions produce a result from data contained in one or more task result documents. In other words, logic expressions operate on **sets** of result documents. The result from the logical expression evaluation is written to one or more new MongoDB result documents (the details will be explained below).

The `NOT` operator requires additional commentary. ClarityNLP supports the use of `NOT` as a synonym for "set difference". Thus `A NOT B` means all elements of set `A` that are NOT also elements of set `B`. The use of `NOT` to mean "set complement" is not supported. Hence expressions such as `NOT A`, `NOT hasRigors`, etc., are invalid NLPQL statements. The `NOT` operator **must** appear between two other expressions.

3. Mixed Expressions

A *mixed* expression is a string containing either:

- A mathematical expression **and** a logic expression
- A mathematical expression using variables involving two or more NLPQL features

For instance:

```
// both math and logic
(Temperature.value >= 100.4) AND (hasDyspnea OR hasTachycardia)

// two NLPQL features: LesionMeasurement and Temperature
(LesionMeasurement.dimension_X >= 10) OR (Temperature.value >= 100.4)
```

(continues on next page)

(continued from previous page)

```
// math, logic, and multiple NLPQL features
Temperature.value >= 100.4 AND (hasRigors OR hasNausea) AND (LesionMeasurement.
↪dimension_X >= 15)
```

The evaluation mechanisms used for mathematical, logic, and mixed expressions are quite different. To fully understand the issues involved, it is helpful to first understand the meaning of the 'intermediate' and 'final' phenotype results.

Phenotype Result CSV Files

Upon submission of a new job, ClarityNLP prints information to stdout that looks similar to this:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 1024
Access-Control-Allow-Origin: *
Server: Werkzeug/0.14.1 Python/3.6.4
Date: Fri, 23 Nov 2018 18:40:38 GMT
{
   "job_id": "11108",
   "phenotype_id": "11020",
   "phenotype_config": "http://localhost:5000/phenotype_id/11020",
   "pipeline_ids": [
        12529,
        12530,
        12531,
        12532,
        12533,
        12534,
        12535
    ],
    "pipeline_configs": [
        "http://localhost:5000/pipeline_id/12529",
        "http://localhost:5000/pipeline_id/12530",
        "http://localhost:5000/pipeline_id/12531",
        "http://localhost:5000/pipeline_id/12532",
        "http://localhost:5000/pipeline_id/12533",
        "http://localhost:5000/pipeline_id/12534",
        "http://localhost:5000/pipeline_id/12535"
    ],
    "status_endpoint": "http://localhost:5000/status/11108",
    "results_viewer": "?job=11108",
    "luigi_task_monitoring": "http://localhost:8082/static/visualiser/index.html
↪#search__search=job=11108",
    "intermediate_results_csv": "http://localhost:5000/job_results/11108/phenotype_
↪intermediate",
    "main_results_csv": "http://localhost:5000/job_results/11108/phenotype"
}
```

Here we see various items relevant to the job submission. Each submission receives a *job_id*, which is a unique numerical identifier for the run. ClarityNLP writes all task results from all jobs to the `phenotype_results` collection in a Mongo database named `nlp`. The job_id is needed to distinguish the data belonging to each run. Results can be extracted directly from the database by issuing MongoDB queries.

We also see URLs for 'intermediate' and 'main' phenotype results. These are convenience APIs that export the results to CSV files. The data in the intermediate result CSV file contains the output from each NLPQL task not marked as

`final`. The main result CSV contains the results from any final tasks or final expression evaluations. The CSV file can be viewed in Excel or in another spreadsheet application.

Each NLP task generates a result document distinguished by a particular value of the `nlpql_feature` field. The *define* statement

```
define hasFever:
     where Temperature.value >= 100.4;
```

generates a set of rows in the intermediate CSV file with the nlpql_feature field set to `hasFever`. The NLP tasks

```
// nlpql_feature 'hasRigors'
define hasRigors:
    Clarity.ProviderAssertion({
        termset: [RigorsTerms],
        documentset: [ProviderNotes]
    });

// nlpql_feature 'hasDyspnea
define hasDyspnea:
    Clarity.ProviderAssertion({
        termset: [DyspneaTerms],
        documentset: [ProviderNotes]
    });
```

generate two blocks of rows in the CSV file, the first block having the nlpql_feature field set to `hasRigors` and the next block having it set to `hasDyspnea`. The different nlpql_feature blocks appear in order as listed in the source NLPQL file. The presence of these nlpql_feature blocks makes locating the results of each NLP task a relatively simple matter.

Expression Evaluation Algorithms

ClarityNLP evaluates expressions via a multi-step procedure. In this section we describe the different processing stages.

Expression Tokenization and Parsing

The NLPQL front end parses the NLPQL file and sends the raw expression text to the evaluator (`nlp/data_access/expr_eval.py`). The evaluator module parses the expression text and converts it to a fully-parenthesized token string. The tokens are separated by whitespace and all operators are replaced by string mnemonics (such as `GE` for the operator >=, `LT` for the operator <, etc.).

If the expression includes any subexpressions involving numeric literals, they are evaluated at this stage and the literal subexpression replaced with the result.

Validity Checks

The evaluator then runs validity checks on each token. If it finds a token that it does not recognize, it tries to resolve it into a series of known NLPQL features separated by logic operators. For instance, if the evaluator were to encounter the token `hasRigorsANDhasDyspnea` under circumstances in which only `hasRigors` and `hasDyspnea` were valid NLPQL features, it would replace this single token with the string `hasRigors AND hasDyspnea`. If it cannot perform the separation (such as with the token `hasRigorsA3NDhasDyspnea`) it reports an error and writes error information into the log file.

If the validity checks pass, the evaluator next determines the expression type. The valid types are `EXPR_TYPE_MATH`, `EXPR_TYPE_LOGIC`, and `EXPR_TYPE_MIXED`. If the expression type cannot be determined, the evaluator reports an error and writes error information into the log file.

Subexpression Substitution

If the expression is of mixed type, the evaluator locates all simple math subexpressions contained within and replaces them with temporary NLPQL feature names, thereby converting math subexpressions to logic subexpressions. The substitution process continues until all mathematical subexpressions have been replaced with substitute NLPQL features, at which point the expression type becomes `EXPR_TYPE_LOGIC`.

To illustrate the substitution process, consider one of the examples from above:

```
Temperature.value >= 100.4 AND (hasRigors OR hasNausea) AND (LesionMeasurement.
↪dimension_X >= 15)
```

This expression is of mixed type, since it contains the mathematical subexpression `Temperature.value >= 100.4`, the logic subexpression `(hasRigors OR hasNausea)`, and the mathematical subexpression `(LesionMeasurement.dimension_X >= 15)`. The NLPQL features in each math subexpression, `Temperature` and `LesionMeasurement`, also differ.

The evaluator identifies the Temperature subexpression and replaces it with a substitute NLPQL feature, `m0` (for instance). This transforms the original expression into:

```
(m0) AND (hasRigors OR hasNausea) AND (LesionMeasurement.dimension_X >= 15)
```

Now only one mathematical subexpression remains.

The evaluator again makes a substitution `m1` for the remaining mathematical subexpression, which converts the original into

```
(m0) AND (hasRigors OR hasNausea) AND (m1)
```

This is now a pure logic expression.

Thus the substitution process transforms the original mixed-type expression into three subexpressions, each of which is of simple math or simple logic type:

```
subexpression 1 (m0): 'Temperature.value >= 100.4'
subexpression 2 (m1): 'LesionMeasurement.dimension_X >= 15'
subexpression 3:      '(m0) AND (hasRigors OR hasNausea) AND (m1)'
```

By evaluating each subexpression in order, the result of evaluating the original mixed-type expression can be obtained.

Evaluation of Mathematical Expressions

Removal of Unnecessary Parentheses

The evaluator next removes all unnecessary pairs of parentheses from the mathematical expression. A pair of parentheses is unnecessary if it can be removed without affecting the result. The evaluator detects changes in the result by converting the expression with a pair of parentheses removed to postfix, then comparing the postfix form with that of the original. If the postfix expressions match, that pair of parentheses was non-essential and can be discarded. The postfix form of the expression has no parentheses, as described below.

Conversion to Explicit Form

After removal of nonessential parentheses, the evaluator rewrites the expression so that the tokens match what's actually stored in the database. This involves an explicit comparison for the NLPQL feature and the unadorned use of the field name for variables. To illustrate, consider the `hasFever` example above:

```
define hasFever:
    where Temperature.value >= 100.4;
```

The expression portion of this define statement is `Temperature.value >= 100.4`. The evaluator rewrites this as:

```
(nlpql_feature == Temperature) AND (value >= 100.4)
```

In this form the tokens match the fields actually stored in the task result documents in MongoDB.

Conversion to Postfix

Direct evaluation of an infix expression is complicated by parenthesization and operator precedence issues. The evaluation process can be greatly simplified by first converting the infix expression to postfix form. Postfix expressions require no parentheses, and a simple stack-based evaluator can be used to evaluate them directly.

Accordingly, a conversion to postifx form takes place next. This conversion process requires an operator precedence table. The NLPQL operator precedence levels match those of Python and are listed here for reference. Lower numbers imply lower precedence, so `or` has a lower precedence than `and`, which has a lower precedence than `+`, etc.

Operator	Precedence Value
(	0
)	0
or	1
and	2
not	3
<	4
<=	4
>	4
>=	4
!=	4
==	4
+	9
-	9
*	10
/	10
%	10
^	12

Conversion from infix to postfix is unambiguous if operator precedence and associativity are known. Operator precedence is given by the table above. All NLPQL operators are left-associative except for exponentiation, which is right-associative. The infix-to-postfix conversion algorithm is the standard one and can be found in the function `_infix_to_postfix` in the file `nlp/data_access/expr_eval.py`.

After conversion to postfix, the `hasFever` expression becomes:

```
'nlpql_feature', 'Temperature', '==', 'value', '100.4', '>=', 'and'
```

Generation of the Aggregation Pipeline

The next task for the evaluator is to convert the expression into a sequence of MongoDB aggregation pipeline stages. This process involves the generation of an initial $match query to filter out everything but the data for the current job. The match query also checks for the existence of all entries in the field list and that they have non-null values. **A simple existence check is not sufficient**, since a null field actually exists but has a value that cannot be used for computation. Hence checks for **existence** and a **non-null value** are both necessary.

For the `hasFever` example, the initial match query generates a pipeline filter stage that looks like this, assuming a job_id of 12345:

```
{
    "$match": {
        "job_id": 12345,
        "nlpql_feature": {"$exists":True, "$ne":None},
        "value"        : {"$exists":True, "$ne":None}
    }
}
```

This match pipeline stage runs first and performs coarse filtering on the data in the result database. It finds only those task result documents matching the specified job_id, and it further restricts consideration to those documents having valid entries for the expression's fields.

Subsequent Pipeline Stages

After generation of the initial match filter stage, the postfix expression is then 'evaluated' by a stack-based mechanism. The result of the evaluation process is **not** the actual expression value, but instead a set of MongoDB aggregation commands that tell MongoDB how to compute the result. The evaluation process essentially generates Python dictionaries that obey the aggregation syntax rules. More information about the aggregation pipeline can be found here.

The pipeline actually does a $project operation and creates a new document with a Boolean field called `value`. This field has a value of True or False according to whether the source document satisfied the mathematical expression. The `_id` field of the projected document matches that of the original, so that a simple query on these `_id` fields can be used to recover the desired documents.

The final aggregation pipeline for our example becomes:

```
// (nlpql_feature == Temperature) and (value >= 100.4)
{
   "$match": {
       "job_id":12345
       "nlpql_feature": {"$exists":True, "$ne":None},
       "value"        : {"$exists":True, "$ne":None}
   }
},
{
    "$project" : {
        "value" : {
            "$and" : [
                {"$eq"  : ["$nlpql_feature", "Temperature"]},
                {"$gte" : ["$value", 100.4]}
            ]
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

The completed aggregation pipeline gets sent to MongoDB for evaluation. Mongo performs the initial filtering operation, applies the subsequent pipeline stages to all surviving documents, and sets the "value" Boolean result. A final query extracts the matching documents and writes new result documents with an `nlpql_feature` field equal to the label from the `define` statement, which for this example would be `hasFever`.

Evaluation of Logic Expressions

The initial stages of the evaluation process for logic expressions proceed similarly to those for mathematical expressions. Unnecessary parentheses are removed and the expression is converted to postfix.

Detection of n-ary AND and OR

After the postfix conversion, a pattern matcher looks for instances of n-ary `AND` and/or `OR` in the set of postfix tokens. An n-ary `OR` would look like this, for n == 4:

```
// infix
hasRigors OR hasDyspnea OR hasTachycardia OR hasNausea

// postfix
hasRigors hasDyspnea OR hasTachycardia OR hasNausea OR
```

The n-value refers to the number of operands. All such n-ary instances are replaced with a variant form of the operator that includes the count. The reason for this is that n-ary `AND` and `OR` can be handled easily by the aggregation pipeline, and their use simplifies the pipeline construction process. For this example, the rewritten postfix form would become:

```
hasRigors hasDyspnea hasTachycardia hasNausea OR4
```

Generation of the Aggregation Pipeline

As with mathematical expressions, the logic expression aggregation pipeline begins with an initial stage that filters on the job_id and checks that the `nlpql_feature` field exists and is non-null. No explicit field checks are needed since logic expressions do not use NLPQL variables. For a job_id of 12345, this inital filter stage is:

```
{
    "$match": {
        "job_id":12345
        "nlpql_feature": {"$exists":True, "$ne":None}
    }
}
```

Following this is another filter stage that removes all docs not having the desired NLPQL features. For the original logic expression example above:

```
hasFever AND (hasDyspnea OR hasTachycardia)
```

this second filter stage would look like this:

```
{
    "$match": {
        "nlpql_feature": {"$in": ['hasFever', 'hasDyspnea', 'hasTachycardia']}
    }
}
```

Grouping by Value of the Context Variable

The next stage in the logic pipeline is to group documents by the **value** of the context field. Recall that NLPQL files specify a context of either 'document' or 'patient', meaning that a document-centric or patient-centric view of the results is desired. In a document context, ClarityNLP needs to examine all data pertaining to a given document. In a patient context, it needs to examine all data pertaining to a given patient.

The grouping operation collects all such data (the ClarityNLP task result documents) that pertain to a given document or a given patient. Documents are distinguished by their `report_id` field, and patients are distinguished by their patient IDs, which are stored in the `subject` field. **You can think of these groups as being the 'evidence' for a given document or for a given patient.** If the patient has the conditions expressed in the NLPQL file, the evidence for it will reside in the group for that patient.

As part of the grouping operation ClarityNLP also generates a **set** of NLPQL features for each group. This set is called the **feature_set** and it will be used to evaluate the expression logic for the group as a whole.

The grouping pipeline stage looks like this:

```
{
    "$group": {
        "_id": "${0}".format(context_field),

        # save only these four fields from each doc; more efficient
        # than saving entire doc, uses less memory
        "ntuple": {
            "$push": {
                "_id": "$_id",
                "nlpql_feature": "$nlpql_feature",
                "subject": "$subject",
                "report_id": "$report_id"
            }
        },
        "feature_set": {"$addToSet": "$nlpql_feature"}
    }
}
```

Here we see the $group operator grouping the documents on the value of the context field. An **ntuple** array is generated for each different value of the context variable. This is the 'evidence' as discussed above. Only the essential fields for each document are used, which reduces memory consumption and improves efficiency. We also see the generation of the feature set for each group, in which each NLPQL feature for the group's documents is added to the set.

At the conclusion of this pipeline stage, each group has two fields: an `ntuple` array that contains the relevant data for each document in the group, and a `feature_set` field that contains the distinct features for the group.

Logic Operation Stage

After the grouping operation, the logic operations of the expression are applied to the elements of the feature set. If a particular patient satisfies the `hasFever` condition, then at least one document in that patient's group will have an

NLPQL feature field with the value of `hasFever`. Since all the distinct values of the NLPQL features for the group are stored in the feature set, the feature set must also have an element equal to `hasFever`.

A check for set membership using aggregation syntax is expressed as:

```
{"$in": ["hasFever", "$feature_set"]}
```

This construct means to use the $in operator to test whether `feature_set` contains the element `hasFever`. The `$in` operator returns a Boolean result.

A successful test for feature set membership means that the patient has the stated feature.

The evaluator implements the expression logic by translating it into a series of set membership tests. For our example above, the logic operation pipeline stage becomes:

```
{
    '$match': {
        '$expr': {
            '$and': [
                {'$in': ['hasFever', '$feature_set']},
                {
                    '$or': [
                        {'$in': ['hasDyspnea', '$feature_set']},
                        {'$in': ['hasTachycardia', '$feature_set']}
                    ]
                }
            ]
        }
    }
}
```

Once again we have a match operation to filter the documents. Only those documents satisfying the expression logic will survive the filter. The $expr operator allows the use of aggregation syntax in contexts where the standard MongoDB query syntax would be required.

Following that we see a series of logic operations for our expression `hasFever AND (hasDyspnea OR hasTachycardia)`. The inner `$or` operation tests the feature set for membership of `hasDyspnea` and `hasTachycardia`. If either or both are present, the `$or` operator returns True. The result of the `$or` is then used in an `$and` operation which tests the feature set for the presence of `hasFever`. If it is also present, the `$and` operator returns True as well, and the document in question survives the filter operation.

To summarize the evaluation process so far: ClarityNLP converts infix logic expressions to postfix form and groups the documents by value of the context variable. It uses a stack-based postfix evaluation mechanism to generate the aggregation statements for the expression logic. Each logic operation is converted to a test for the presence of an NLPQL feature in the feature set.

Final Aggregation Pipeline

With these operations the pipeline is complete. The full pipeline for our example is:

```
// aggregation pipeline for hasFever AND (hasDyspnea OR hasTachycardia)

// filter documents on job_id and check validity of the nlpql_feature field
{
    "$match": {
        "job_id":12345
        "nlpql_feature": {"$exists":True, "$ne":None}
```

(continues on next page)

(continued from previous page)

```
    }
},

// filter docs on the desired NLPQL feature values
{
    "$match": {
        "nlpql_feature": {"$in": ['hasFever', 'hasDyspnea', 'hasTachycardia']}
    }
},

// group docs by value of context variable and create feature set
{
    "$group": {
        "_id": "${0}".format(context_field),
        "ntuple": {
            "$push": {
                "_id": "$_id",
                "nlpql_feature": "$nlpql_feature",
                "subject": "$subject",
                "report_id": "$report_id"
            }
        },
        "feature_set": {"$addToSet": "$nlpql_feature"}
    }
},

// perform expression logic on the feature set
{
    '$match': {
        '$expr': {
            '$and': [
                {'$in': ['hasFever', '$feature_set']},
                {
                    '$or': [
                        {'$in': ['hasDyspnea', '$feature_set']},
                        {'$in': ['hasTachycardia', '$feature_set']}
                    ]
                }
            ]
        }
    }
}
```

Result Generation

After constructing a math or logic aggregation pipeline, the evaluator runs the pipeline and receives the results from MongoDB. The result set is either a list of document ObjectID values (`_id`) for a math expression or an ObjectId list with group info for logic expressions. For math expressions, the documents whose `_id` values appear in the list are queried and written out as the result set. These documents have their `nlpql_feature` field set to that of the `define` statement that contained the expression.

For logic expressions the process is more complex. To help explain what the evaluator does we present here a representation of the grouped documents after running the pipeline above, for the expression `hasFever AND (hasDyspnea OR hasTachycardia)`:

ObjectId (_id)	nlpql_feature	subject	report_id
5c2e9e3431ab5b05db3430e1	hasDyspnea	19054	798209
5c2e9e3431ab5b05db3430e2	hasDyspnea	19054	798209
5c2e9e3431ab5b05db3430e3	hasDyspnea	19054	798209
5c2e9e3431ab5b05db3430e4	hasDyspnea	19054	798209
5c2e9ec931ab5b05db343efa	hasDyspnea	19054	1303796
5c2ea2bd31ab5b05db34868c	hasTachycardia	19054	1699977
5c2ea2bd31ab5b05db34868d	hasTachycardia	19054	1699977
5c2ea35a31ab5b05db348f19	hasTachycardia	19054	1802359
5c2ea3a531ab5b05db3492f6	hasTachycardia	19054	1905337
5c2ea42431ab5b05db34998c	hasTachycardia	19054	1802375
5c2ea42431ab5b05db34998d	hasTachycardia	19054	1802375
5c2eb55831ab5b05db35097b	hasFever	19054	['1264178']
5c2eb55831ab5b05db350d45	hasFever	19054	['1699944']
5c2eb55831ab5b05db350d46	hasFever	19054	['1699944']

Here we see a representation of the document group for patient 19054. This group of documents can be considered to be the "evidence" for this patient. In the ObjectID column are the MongoDB ObjectID values for each task result document or mathematical result document. The `nlpql_feature` column shows which NLPQL feature ClarityNLP found for that document. The `subject` column shows that all documents in the group belong to patient 19054, and the `report_id` column shows the document identifier.

We see that patient 19054 has five instances of `hasDyspnea`, six instances of `hasTachycardia`, and three instances of `hasFever`. You can consider this group as being composed of three subgroups with five, six, and three elements each.

ClarityNLP presents result documents in a "flattened" format. For each NLPQL label introduced in a "define" statement, ClarityNLP generates a set of result documents containing that label in the `nlpql_feature` field. Each result document also contains a record of the source documents that were used as evidence for that label.

Flattening of the Result Group

To flatten these results and generate a set of output documents labeled by the `hasSymptoms` NLPQL feature (from the original "define" statement), ClarityNLP essentially has two options:

- generate **all possible ways** to derive `hasSymptoms` from this data
- generate the **minimum number of ways** to derive `hasSymptoms` from this data (while not ignoring any data)

The **maximal** result set can be generated by the following reasoning. First, in how many ways can patient 19054 satisfy the condition `hasDyspnea OR hasTachycardia`? From the data in the table, there are five ways to satisfy the `hasDyspnea` condition and six ways to satisfy the `hasTachycardia` condition, for a total of 5 + 6 = 11 ways. Then, for **each** of these ways, there are three ways for the patient to satisfy the condition `hasFever`. Thus there are a total of 3 * (5 + 6) = 3 * 11 = 33 ways for this patient to satisfy the condition `hasFever AND (hasDyspnea OR hasTachycardia)`, which would result in the generation of 33 output documents under a maximal representation.

The **minimal** result set can be generated by the following reasoning. We have seen that there are 11 ways for this patient to satisfy the condition `hasDyspnea OR hasTachycardia`. Each of these must be paired with a `hasFever`, from the logical `AND` operator in the expression. By repeating each of the `hasFever` entries, we can "tile" the output and pair a `hasFever` with one of the 11 others. This procedure generates a result set containing only 11 entries instead of 33. It uses all of the output data, and it **minimizes** data redundancy.

In general, the cardinalities of the sets of NLPQL features connected by logical `OR` are added together to compute the number of possible results. For features connected by logical `AND`, the cardinalities are multiplied to get the total num-

ber of possiblilities under a maximal representation (this is the Cartesian product). Under a minimal representation, the cardinality of the result is equal to the maximum cardinality of the constitutent subsets.

So which output representation does ClarityNLP use?

ClarityNLP uses the minimal representation of the output data.

Here is what the result set looks like using a minimal representation. Each of the 11 elements contains a pair of documents, one with the feature `hasFever` and the other having either `hasDyspnea` or `hasTachycardia`, as required by the expression. We show only the last four hex digits of the ObjectID for clarity:

```
// expression: hasFever AND (hasDyspnea OR hasTachycardia)

('097b', 'hasFever'), ('30e1', 'hasDyspnea')
('0d45', 'hasFever'), ('30e2', 'hasDyspnea')
('0d46', 'hasFever'), ('30e3', 'hasDyspnea')
('097b', 'hasFever'), ('30e4', 'hasDyspnea')
('0d45', 'hasFever'), ('3efa', 'hasDyspnea')
('0d46', 'hasFever'), ('868c', 'hasTachycardia')
('097b', 'hasFever'), ('868d', 'hasTachycardia')
('0d45', 'hasFever'), ('8f19', 'hasTachycardia')
('0d46', 'hasFever'), ('92f6', 'hasTachycardia')
('097b', 'hasFever'), ('998c', 'hasTachycardia')
('0d45', 'hasFever'), ('998d', 'hasTachycardia')
```

Note that the three `hasFever` entries repeat three times, followed by another repeat of the first two entries to make a total of 11. Each of these is paired with one of the five `hasDyspnea` entries or one of the six `hasTachycardia` entries. No data for this patient has been lost, and the result is 11 documents in a flattened format satisfying the logic of the original expression.

Testing the Expression Evaluator

There is a comprehensive test program for the expression evaluator in the file `nlp/data/access/expr_tester.py`. The test program requires a running instance of MongoDB. We strongly recommend running Mongo on the same machine as the test program to minimize data transfer delays.

The test program loads a data file into MongoDB and evaluates a suite of expressions using the data. The expression logic is separately evaluated with Python set operations. The results from the two evaluations are compared and the tests pass only if both evaluations produce identical sets of patients.

The test program can be run from the command line. For usage info, run with the `--help` option:

```
python3 ./expr_tester.py --help
```

The test program assumes that the user has permission create a database without authentication.

To run the test suite with the default options, first launch MongoDB on your local system. Information about how to do that can be found in our *native setup guide*.

After MongoDB initializes, run the test program with this command, assuming the default Mongo port of 27017:

```
python3 ./expr_tester.py
```

If your MongoDB instance is hosted elsewhere or uses a non-default port number, provide the connection parameters explicitly:

```
python3 ./expr_tester.py --mongohost <ip_address> --mongoport <port_number>
```

The test program takes several minutes to run. Upon completion it should report that all tests passed.

Building Custom Task Algorithms

Custom Task Algorithms

Building custom task algorithms in ClarityNLP is a way to create custom algorithms and include external libraries that are callable from NLPQL. To begin creating custom task algorithms, you need a few things to get started.

Create a Python Class

In `nlp/custom_tasks`, create a new Python class that extends `BaseTask`. `BaseTask` is a class that sets up the data, context and connections needed to read and write data in ClarityNLP. See the source code for `BaseTask` at `nlp/tasks/task_utilities.py`. You can start with the sample below and copy and paste the basic structure that you need for a custom task.

```
from tasks.task_utilities import BaseTask
from pymongo import MongoClient

class SampleTask(BaseTask):
    task_name = "MyCustomTask"

    def run_custom_task(self, temp_file, mongo_client: MongoClient):
        for doc in self.docs:

            # you can get sentences and text through these utility methods
            text = self.get_document_text(doc)
            sentences = self.get_document_sentences(doc)

            # put your custom algorithm here, save your output to a dictionary, and␣
↪write results below
            obj = {
                'foo': 'bar',
                'sentence_one': sentences[0]
            }

            # writing results
            self.write_result_data(temp_file, mongo_client, doc, obj)

            # writing to log (optional)
            self.write_log_data("DONE", "done writing sample data")
```

Task Name

`task_name` is important to include in your SampleTask if you want a user-friendly name to be called from NLPQL. In this example, the task name is `MyCustomTask`, but if a custom task wasn't specified, the task name would be `SampleTask`. Also, it's important to be aware with naming that you can overwrite other custom tasks and even core ClarityNLP tasks (which may be the desired outcome). So in most cases, you'll want to provide a unique name.

Running a Custom Task

The ClarityNLP engine will automatically create a distributed job and assign a set of documents to each worker task. Knowing that, there are just a few things to do to create custom tasks. You'll need to implement the `run_custom_task` function in your task. That will give you access to the `self` parameter which has attributes

from the job and the set of documents your algorithm will run on. You don't need to worry about them too much, but know they are accessible in your custom task.

You also have access to a `temp_file` which is provided by Luigi. It's not necessarily used by ClarityNLP, but you may wish to use it for some logging purpose (logging will be discussed more below). In addition, you have a `mongo_client` connection that is opened and closed for you, however you'll need access to this object when you're writing output for your ClarityNLP NLPQL.

Iterating over Documents

Since you are responsible for a set of documents, you need to get the list of documents which has been assigned to this worker. This is callable by using `self.docs` and should be iterable in Python.

Per document (or `doc`), there are few helper functions available for you.

- `self.get_document_text(doc)` - gets the text of document as a string
- `self.get_document_sentences(doc)` - gets a list of the sentences in a document, parsed with the default ClarityNLP sentence segmentor

Accessing Custom Variables

If you have custom parameters in your NLPQL, you can access them via the `custom_arguments` dictionary in your pipeline config.

```
my_value = self.pipeline_config.custom_arguments['my_value']
```

Saving Results

All data that ClarityNLP uses in NLPQL needs to eventually end up in MongoDB. `BaseTask` provides two types of hooks, depending on whether you have a single object or a list of objects. Both return the new unique id (or ids) from MongoDB.

- `self.write_result_data(temp_file, mongo_client, doc, obj)` - saves results where `obj` is a Python `dict`
- `self.write_multiple_result_data(temp_file, mongo_client, doc, obj_list)` - saves results where `obj_list` is a Python `list` or `set` (implies multiple results per document)

Logging and Debugging

ClarityNLP provides two means for logging and debugging your custom tasks. Most commonly you will use the first method, where you pass in a **status** and **description text**. This is written to the Postgres database, and accessible when users call the status function on their NLPQL jobs.

```
self.write_log_data("DONE!", "done writing sample data")
```

The second is less common, but may be desirable in certain cases, which is writing to the `temp_file` used by Luigi, e.g.:

```
temp_file.write("Some pretty long message that maybe I don't want to show to users")
```

This is written to the file system and generally not accessible to users via APIs.

Using Custom Collectors

Collectors in ClarityNLP are similar to the reduce step in map-reduce jobs. They can be implemented similar to custom tasks, except their purpose is generally to summarize across all the data generated in the parallelized Luigi tasks. To utilize the collector, extend the `BaseCollector` class, and make sure your `collector_name` in that class is the same as your `task_name` in your custom task.

```
class MyCustomCollector(BaseCollector):
    collector_name = 'cool_custom_stuff'

    def custom_cleanup(self, pipeline_id, job, owner, pipeline_type, pipeline_config,␣
↪client, db):
        print('custom cleanup (optional)')

    def run_custom_task(self, pipeline_id, job, owner, pipeline_type, pipeline_config,
↪ client, db):
        print('run custom task collector')
        # TODO write out some summary stats to mongodb


class MyCustomTask(BaseTask):
    task_name = 'cool_custom_stuff'

    def run_custom_task(self, temp_file, mongo_client: MongoClient):
        print('run custom task')

        for doc in self.docs:
            # TODO write out some data to mongodb about these docs
```

Collectors often are not needed, but may be necessary for certain algorithm implementations.

Setting up the Python Package

ClarityNLP automatically discovers any classes in the `custom_task` package. However, besides saving your Python file in `custom_tasks`, you just need to make sure it's included in the `custom_tasks` package by adding it to `nlp/custom_tasks/__init__.py`, following the example:

```
from .SampleTask import SampleTask
```

Calling Custom Algorithms from NLPQL

To run your custom algorithm in NLPQL, you just need to call it by name as a function like the example below, and make sure to pass in any variables needed for the config and Solr query.

```
define sampleTask:
  Clarity.MyCustomTask({
    documentset: [ProviderNotes],
    "my_custom_argument": 42
  });
```

Custom Algorithm or External Library?

There aren't too many limitations on what you build inside of custom tasks and collectors, given that it's a something that can input text, and output a Python object. This is a powerful feature that will allow you to integrate many types of capabilities into ClarityNLP!

Other Conventions

While the previous sections contain the main items you need to create custom task algorithms in ClarityNLP, here's some other information that might be useful.

- **Default Value**: Or using `value` as the default field. In NLPQL, when no field name is specified, it will default to `value`. This means that you may want to provide a `value` field in your resulting object that gets saved to MongoDB, so that there's a default value
- **Sentences**: While there's no requirement to parse or run your algorithm at the sentence level, it is useful for scoping and user validation. Therefore, in most of the core ClarityNLP algorithms, output `sentence` is part of the result, and you may wish to follow this paradigm
- **Metadata**: All the metadata from the job is automatically saved for you, however you may have additional metadata you want to save from your algorithm or source data

Testing

Testing NLP Algorithms

This application uses *pytest*.

Running Pytest from the *nlp* directory

From the command line:

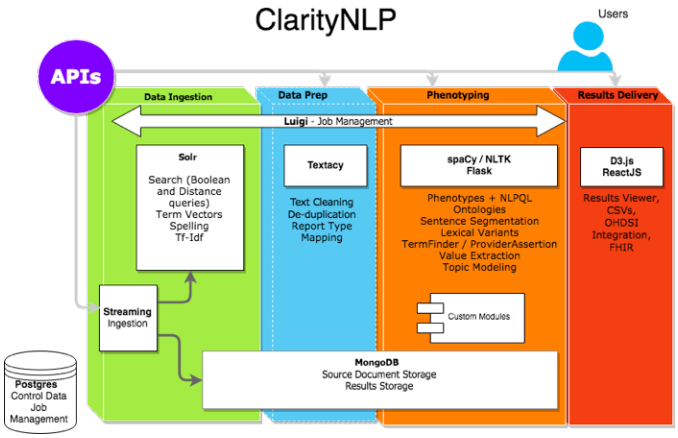
```
python3 -m pytest tests/
```

1.3.2 For App Developers

ClarityNLP Architecture

This library uses Python 3.6+. The source code is hosted here.

Here's an overview of ClarityNLP's architecture.



Third-Party App Integration

Third Party App Integration

The information below will help you configure a third-party application for ClarityNLP.

By "third-party", we are referring to applications not developed by the core ClarityNLP team. The third-party app would like access to your ClarityNLP's resources.

The third-party application must be a registered OAuth2 Client with ClarityNLP's Identity Provider in order to complete an OAuth2 Flow and be issued an access token.

If you need a refresher on OAuth2 in order to determine the ideal Grant Type for the third-party application, here is a review.

Once you have determined the appropriate Grand Type, refer to `/identity-provider/Config.cs` to see examples of how to configure your client.

An exhaustive list of Client properties can be found here.

FHIR Integration

FHIR Integration

1.4 IT Guide

This portion of the guide is primarily for system administrators or information technology support personnel.

1.4.1 System Architecture

System Architecture

1.4.2 Networking

Networking

1.4.3 Security

Security

1.5 NLPQL Reference

1.5.1 NLPQL Helpers

documentset

ClarityNLP modules in NLPQL that defines how documents are to be queried in Solr.

Functions

Clarity.createReportTagList

Uses the ClarityNLP document ontology. Mapped using the Report Type Mapper.

```
documentset RadiologyNotes:
    Clarity.createReportTagList(["Radiology"]);
```

Clarity.createDocumentSet

Uses arguments to build a custom Solr query to retrieve document set. All arguments are optional, but at least one must be present.

Name	Type	Notes
report_types	List[str]	List of report types. Corresponds to *report_types* in Solr.
report_tags	List[str]	List of report tags. Report tags mapped to document ontology.
source	str OR List[str]	List of sources to map to. Use array of strings or string, separated by commas.
filter_query	str	Use single quote (‘) to quote. Corresponds to Solr *fq* parameter. See here.*
query	str	Use single quote (‘) to quote. Corresponds to Solr *q* parameter. See here.*

* See more about the ClarityNLP Solr fields here.

```
documentset AmoxDischargeNotes:
 Clarity.createDocumentSet({
     "report_types":["Discharge summary"],
     "report_tags": [],
     "filter_query": "",
     "source": ["MIMIC","FDA Drug Labels"],
     "query":"report_text:amoxicillin"});
```

Clarity.createReportTypeList

Uses an explicit report type list of string to match from the *report_type* field.

```
documentset ChestXRDocs:
    Clarity.createReportTypeList(["CHEST XR", "CHEST X-RAY"]);
```

---

cohort

Limits Solr query patients by ones matching the cohort.

Functions

OHDSI.getCohort(cohortId)

Returns a list of patients matching the OHDSI cohort id. Will limit patients in the Solr query.

```
cohort SocialSupportPatients:OHDSI.getCohort(100);
```

cohort can then be passed as an argument in tasks. For example:

```
define Widowed:
    Clarity.ProviderAssertion({
        cohort:SocialSupportPatients,
        termset:[WidowedTerms]
    });
```

Clarity.getJobResults

Returns a list of patients or documents, matching the job_id and parameters. Will limit patients or documents in the Solr query.

Example:

```
cohort OpiodPatients:
    Clarity.getJobResults({
        "context":"patient",
        "job_id": 406,
        "nlpql_feature":"tookOpioids"
    });
```

Arguments:

Name	Type	Required	Notes
context	str	Yes	"patient" or "document"
job_id	int	No	The job_id. Not strictly required, but desirable to select the correct phenotype.
nlpql_feature	str	No	The feature name used in the NLPQL *define*
report_type	str	No	
pipeline_type	str	No	The NLPQL pipeline feature type (e.g. "ValueExtractor")
pipeline_id	int	No	
subject	str	No	
phenotype_final	bool	No	Whether the results were tagged as *final* or not
<any_generated_feature><type>		No	Any feature you wish to filter or that was generated by ClarityNLP.

termset

ClarityNLP modules in NLPQL that define sets of terms.

Example:

```
termset EjectionFractionTerms: [
    "ejection fraction",
    "LVEF",
    "EF",
];
```

termset can now be passed as an argument to tasks. For example:

```
define EjectionFractionFunction:
    Clarity.ValueExtraction({
        termset:[EjectionFractionTerms],
        documentset: [ProviderNotes],
        });
```

Note that *termset* is required in certain tasks such as *Clarity.ProviderAssertion* and *Clarity.TermFinder*.

Lexical Variants

As an optional step, NLPQL can be pre-processed with *lexical variants*. Learn more about how to use lexical variants *here*.

context

Optional field. Required if doing logical operations. Should logical joins occur internally within a document (*Document*), or across all documents for a patient/subject (*Patient*). The default value is *Patient*.

```
context Patient;
```

Termset Expansion Macros

NLPQL supports a set of macros for termset generation. The macros provide a compact syntax for representing lists of synonyms and lexical variants (plurals and verb inflections). The macros also support the concept of a "namespace", so that terms can be generated from different sources.

The use of termset expansion macros is **optional**. They are provided purely for convenience, as a means to generate and suggest additional synonyms.

Syntax

The macro syntax is `namespace.function(args)`, where the namespace is either `Clarity` or `OHDSI`. The argument is either a single term in double quotes or a comma-separated list of terms surrounded by brackets:

```
namespace.function("term")
namespace.function(["term1", "term2", ..., "termN"])
```

If the namespace is omitted it defaults to `Clarity`. The supported macros are:

Macro Meaning	
Clarity.Synonyms	Generate a list of synonyms from WordNet
Clarity.Plurals	Generate a list of plural forms
Clarity.VerbInflections	Generate inflections for the verb in **base** form
OHDSI.Synonyms	Generate a list of OHDSI synonyms for the concept
OHDSI.Ancestors	Generate all OHDSI ancestor concepts
OHDSI.Descendants	Generate all OHDSI descendant concepts

The synonym finder examines the macro argument(s) and attempts to find the nouns, adjectives, and adverbs. It generates synonyms for each that it finds, returning the cartesian product[1] of all possibilities. This process can cause a combinatorial explosion in the number of results. To illustrate, consider this example:

```
The human walks the pet.
```

If the synonyms for `human` are `man, woman, boy, girl` and the synonyms for `pet` are `dog, cat`, then 4*2 = 8 results will be generated, in addition to the original:

```
The human walks the pet.
The man walks the dog.
The woman walks the dog.
The boy walks the dog.
The girl walks the dog.
The man walks the cat.
The woman walks the cat.
The boy walks the cat.
The girl walks the cat.
```

Hundreds or perhaps thousands of result strings could be generated by expansion of terms with many synonyms. So we recommend caution with synonym generation, limiting its use to single terms or short strings.

Both single and multiword terms can be included in a macro, and the macro can operate only on selected terms in a list:

[1] https://en.wikipedia.org/wiki/Cartesian_product

```
Synonyms(["heart", "heart attack", "heart disease"])
"heart", Synonyms("heart attack"), "heart disease",
```

IMPORTANT NOTE: the `VerbInflections` macro requires that the verb be given in base form (also called "raw infinitive" form, "dictionary" form, or "bare" form). The reason for this is because it is not possible to unambiguously determine the base form of a verb from an arbitrary inflection, and the ClarityNLP verb inflector requires the base form as input. See the documentation for the *verb inflector* for more on this topic.

Macro Nesting

Macros can also be nested:

```
Clarity.LexicalVariants(OHDSI.Synonyms(["myocardial infarction"]))
Plurals(Synonyms("neoplasm"))
```

The nesting depth is limited to two, as these examples illustrate.

API

The API endpoint `nlpql_expander` allows users to view the results of macro expansion. For instance, to expand macros in the NLPQL file `macros.nlpql`, HTTP POST the file to the `nlpql_expander` API endpoint with this cURL[2] command:

```
curl -i -X POST http://localhost:5000/nlpql_expander -H "Content-Type: text/plain" --
↪data-binary "@macros.nlpql"
```

Another HTTP client, such as Postman[3], could also be used to POST the file.

Examples

Here is an example that illustrates the use of the NLPQL macros.

Consider this termset for symptoms related to influenza:

```
termset FluTermset: [
"coughing",
OHDSI.Synonyms("fever"),
Synonyms("body ache"),
VerbInflections("have fever"),
];
```

After macro expansion, the termset becomes:

```
termset FluTermset: [
"coughing",
"febrile", "fever", "fever (finding)", "pyrexia", "pyrexial",
"body ache", "body aching", ... "torso aching", "trunk ache", "trunk aching",
"had fever", "has fever", "have fever", "having fever",
];
```

[2] https://curl.haxx.se/
[3] https://www.getpostman.com/

Some synonyms for "body ache" have been omitted. The result will obviously require editing and removal of irrelevant synonyms. One could use the macros as part of an iterative development process for termsets, using the macros to generate initial lists of terms which would then be pruned and refined.

References

1.5.2 NLPQL Tasks

All tasks (or data entities) are prefixed in NLPQL as *define*, with the optional *final* flag. The *final* flag writes each result as part of the finalized result set in MongoDB.

Core Tasks

Clarity.MeasurementFinder

Description

Task for extracting size measurements from text, based on the given *termset*. Read more about MeasurementFinder *here*.

Example

```
define ProstateVolumeMeasurement:
    Clarity.MeasurementFinder({
        documentset: [RadiologyReports],
        termset: [ProstateTerms]
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	Yes	
documentset	*documentset*	No	
cohort	*cohort*	No	
sections	List[str]	No	Limit terms to specific sections

Results

Name	Type	Notes
sentence	str	Sentence where measurement is found
text	str	text of the complete measurement
start	int	offset of the first character in the matching text
end	int	offset of the final character in the matching text plus 1
value	str	numeric value of first number (same as *dimension_X*)
term	str	term from *termset* that matched a measurement
dimension_X	int	numeric value of first number
dimension_Y	int	numeric value of second number
dimension_Z	int	numeric value of third number
units	str	either mm, mm2, or mm3
location	List[str]	location of measurement, if detected
condition	str	either 'RANGE' for numeric ranges, or 'EQUAL' for all others
temporality	str	CURRENT or PREVIOUS, indicating when the measurement occurred
min_value	int	either *min([x, y, z])* or *min(values)*
max_value	int	either *max([x, y, z])* or *max(values)*

Collector

No

Clarity.NamedEntityRecognition

Description

Simple task that runs spaCy's NER model.

Example

```
Clarity.NamedEntityRecognition({
  documentset: [FDANotes]
});
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	
sections	List[str]	No	Limit terms to specific sections

Results

Name	Type	Notes
term	str	The original entity text.
text	str	Same as *term*
start	int	Index of start of entity
end	int	Index of end of entity
label	str	Label of the entity, e.g. PERSON, MONEY, DATE. See here for more
description	str	Description of the entity

Collector

No

Clarity.ngram

Description

Task that aggregates n-grams across the selected document set. Uses textacy. There's no need to specify *final* on this task. Any n-gram that occurs at at least the minimum frequency will show up in the final result.

Example

```
define demographicsNgram:
  Clarity.ngram({
    termset:[DemographicTerms],
    "n": "3",
    "filter_nums": false,
    "filter_stops": false,
    "filter_punct": true,
    "min_freq": 2,
    "lemmas": true,
    "limit_to_termset": true
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	
n	int	No	Default = 2
filter_nums	bool	No	Default = false; Exclude numbers from n-grams
filter_stops	bool	No	Default = true; Exclude stop words
filter_punct	bool	No	Default = true; Exclude punctuation
lemmas	bool	No	Default = true; Converts work tokens to lemmas
limit_to_termset	bool	No	Default = false; Only include n-grams that contain at least one term from *termset*
min_freq	bool	No	Default = 1; Minimum frequency for n-gram to return in final result

Results

Name	Type	Notes
text	str	The n-gram detected
count	int	The number of occurrences of the n-gram

Collector

BaseCollector

Clarity.POSTagger

Description

Simple task that runs spaCy's Part of Speech Tagger. Should not be ran on large data sets, as will result in a lot of data generation.

Example

```
Clarity.POSTagger({
  documentset: [FDANotes]
});
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

Name	Type	Notes
sentence	str	
term	str	Token being evaluated
text	str	Same as *term*
lemma	str	Lemma of term
pos	str	POS tag. See list here.
tag	str	extended part-of-speech tag
dep	str	dependency label
shape	str	Token shape
is_alpha	bool	Is token all alphabetic
is_stop	bool	Is token a stop word
description	str	Tag description

Collector

No

Clarity.ProviderAssertion

Description

Simple task for identifying positive terms that are not hypothethical and related to the subject. Read more *here*.

Example

```
Clarity.ProviderAssertion({
  cohort:RBCTransfusionPatients,
  termset: [PRBCTerms],
  documentset: [ProviderNotes]
});
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	Yes	
documentset	*documentset*	No	
cohort	*cohort*	No	
sections	List[str]	No	Limit terms to specific sections
include_synonyms	bool	No	
include_descendants	bool	No	
include_ancestors	bool	No	
vocabulary	str	No	Default: 'MIMIC'

Results

Name	Type	Notes
sentence	str	Sentence where the term was found.
section	str	Section where the term was found.
term	str	Term identified
start	str	Start position of term in sentence.
end	str	End position of term in sentence.
negation	str	Negation identified by ConText.
temporality	str	Temporality identified by ConText.
experiencer	str	Experiencer identified by ConText.

Collector

No

Clarity.TermProximityTask

Description

This is a custom task for performing a term proximity search. It takes two lists of search terms and a maximum word distance. If terms from lists 1 and 2 both appear in the sentence and are within the specified distance, the search succeeds and both terms appear in the results. A boolean parameter can also be provided that either enforces or ignores the order of the terms.

Example

```
define final TermProximityFunction:
    Clarity.TermProximityTask({
        documentset: [Docs],
        "termset1": [ProstateTerms],
        "termset2": "cancer, Gleason, Gleason's, Gleasons",
        "word_distance": 5,
        "any_order": "False"
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
documentset	*documentset*	No	
cohort	*cohort*	No	
termset1	*termset* or str	Yes	*termset* or comma-separated list of terms to search for
termset2	*termset* or str	Yes	*termset* or comma-separated list of terms to search for
word_distance	int	Yes	max distance between search terms
any_order	bool	No	Default = false; Should terms in set1 come before terms in set1?

Results

Name	Type	Notes
sentence	str	
start	int	Start of entire matched phrase
end	int	End of entire matched phrase
value	str	Comma separated list of matched terms
word1	str	First term matched
word2	str	Second term matched
start1	int	Start of first term
start2	int	End of second term
end1	int	Start of first term
end2	int	End of second term

Collector

No

Clarity.TermFinder

Description

Simple task for identifying terms with their sections, negation, temporality and experiencer. Read more *here*.

Example

```
Clarity.TermFinder({
  cohort:RBCTransfusionPatients,
  termset: [PRBCTerms],
  documentset: [ProviderNotes]
});
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	Yes	
documentset	*documentset*	No	
cohort	*cohort*	No	
sections	List[str]	No	Limit terms to specific sections
include_synonyms	bool	No	
include_descendants	bool	No	
include_ancestors	bool	No	
vocabulary	str	No	Default: 'MIMIC'

Results

Name	Type	Notes
sentence	str	Sentence where the term was found.
section	str	Section where the term was found.
term	str	Term identified
start	str	Start position of term in sentence.
end	str	End position of term in sentence.
negation	str	Negation identified by ConText.
temporality	str	Temporality identified by ConText.
experiencer	str	Experiencer identified by ConText.

Collector

No

Clarity.ValueExtraction

Description

Extract values from text, related to terms. Read more *here*.

Examples

```
define NYHAClass:
  Clarity.ValueExtraction({
    termset:[NYHATerms],
    enum_list:   ["ii","iii","iv"];
    });
```

```
define Temperature:
  Clarity.ValueExtraction({
    cohort:PlateletTransfusionPatients,
    termset:[TempTerms],
    minimum_value: "96",
    maximum_value: "106"
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	Yes	List of possible terms to find, e.g. 'NYHA'
documentset	*documentset*	No	
cohort	*cohort*	No	
enum_list	List[str]	No	List of possible values to find
minimum_value	int	No	Minimum possible value
maximum_value	int	No	Maximum possible value
case_sensitive	bool	No	Default = false; Is value case sensitive

Results

Name	Type	Notes
sentence	str	Sentence where measurement is found
text	str	text of the complete measurement
start	int	offset of the first character in the matching text
end	int	offset of the final character in the matching text plus 1
value	str	numeric value of first number (same as *dimension_X*)
term	str	term from *termset* that matched a measurement
dimension_X	int	numeric value of first number
dimension_Y	int	numeric value of second number
dimension_Z	int	numeric value of third number
units	str	either mm, mm2, or mm3
location	List[str]	location of measurement, if detected
condition	str	either 'RANGE' for numeric ranges, or 'EQUAL' for all others
temporality	str	CURRENT or PREVIOUS, indicating when the measurement occurred
min_value	int	either *min([x, y, z])* or *min(values)*
max_value	int	either *max([x, y, z])* or *max(values)*

Collector

No

Custom Tasks

Clarity.CQLExecutionTask

Description

This is a custom task that allows ClarityNLP to execute CQL (Clinical Quality Language) queries embedded in NLPQL files. ClarityNLP directs CQL code to a FHIR (Fast Healthcare Interoperability Resources) server, which runs the query and retrieves structured data for a **single** patient. The data returned from the CQL query appears in the job results for the NLPQL file.

The CQL query requires several FHIR-related parameters, such as the patient ID, the URL of the FHIR server, and several others to be described below. These parameters can either be specified in the NLPQL file itself or supplied by ClarityNLP as a Service.

Documentsets for Unstructured and Structured Data

ClarityNLP was originally designed to process *unstructured* text documents. In a typical workflow the user specifies a *documentset* in an NLPQL file, along with the tasks and NLPQL expressions needed to process the documents. ClarityNLP issues a Solr query to retrieve the matching documents, which it divides into batches. ClarityNLP launches a separate task per batch to process the documents in parallel. The number of tasks spawned by the Luigi scheduler depends on the number of *unstructured* documents returned by the Solr query. In general, the results obtained include data from multiple patients.

ClarityNLP can also support single-patient structured CQL queries with a few simple modifications to the documentset. For CQL queries the documentset must be specified in the NLPQL file so that it limits the unstructured documents to those for a **single** patient only. FHIR is essentially a single-patient readonly data retrieval standard. Each patient with data stored on a FHIR server has a unique patient ID. This ID must be used in the documentset statement and in the `Clarity.CQLExecutionTask` body itself, as illustrated below. The documentset specifies the unstructured data for the patient, and the CQL query specifies the structured data for the patient.

Relevant FHIR Parameters

These parameters are needed to connect to the FHIR server, evaluate the CQL statements, and retrieve the results. They can be provided directly as parameters in the `CQLExecutionTask` statement (see below), or indirectly via ClarityNLPaaS:

Parameter	Meaning
cql_eval_url	URL of the FHIR server's CQL Execution Service
patient_id	Unique ID of patient whose data will be accessed
fhir_data_service_uri	FHIR service base URL
fhir_terminology_service_endpoint	Set to `Terminology Service Endpoint`
fhir_terminology_service_uri	URI for a service that conforms to the FHIR Terminology Service Capability Statement
fhir_terminology_user_name	Username for terminology service authentication
fhir_terminology_user_password	Password for terminology service authentication

The terminology user name and password parameters may not be required, depending on whether or not the terminology server enforces password authentication.

Time Filtering

This task supports a time filtering capability for the CQL query results. Two **optional** parameters, `time_start` and `time_end`, can be used to specify a time window. Any results whose timestamps lie *outside* of this window will be discarded. If the time window parameters are omitted, all results from the CQL query will be kept.

The `time_start` and `time_end` parameters must be quoted strings with syntax as follows:

```
DATETIME(YYYY, MM, DD, HH, mm, ss)
DATE(YYYY, MM, DD)
EARLIEST()
LATEST()
```

An optional offset in days can be added or subtracted to these:

```
LATEST() - 7d
DATE(2010, 7, 15) + 20d
```

The offset consists of digits followed by a `d` character, indicating days.

Both ‘‘time_start‘‘ and ‘‘time_end‘‘ are assumed to be expressed in Universal Coordinated Time (UTC).

Here are some time window examples:

1. Discard any results not occurring in March, 2016:

```
"time_start":"DATE(2016, 03, 01)",
  "time_end":"DATE(2016, 03, 31)"
```

2. Keep all results within one week of the most recent result:

```
"time_start":"LATEST() - 7d",
  "time_end":"LATEST()"
```

3. Keep all results within a window of 20 days beginning July 4, 2018, at 3 PM:

```
"time_start":"DATETIME(2018, 7, 4, 15, 0, 0)",
  "time_end":"DATETIME(2018, 7, 4, 15, 0, 0) + 20d"
```

Note that the strings to the left and right of the colon must be surrounded by quotes.

Example

Here is an example of how to use the `CQLExecutionTask` directly, *without* using ClarityNLPaaS. In the text box below there is a documentset creation statement followed by an invocation of the `CQLExecutionTask`. The documentset consists of all indexed documents for patient `99999` with a `source` field equal to `MYDOCS`. These documents are specified explicitly in the `CQLExecutionTask` invocation that follows, to limit the source documents to those for patient 99999 only.

The `task_index` parameter is used in an interprocess communication scheme for controlling task execution. ClarityNLP's Luigi scheduler creates worker task clones in proportion to the number of *unstructured* documents in the documentset. Only a single task from among the clones should actually connect to the FHIR server, run the CQL query, and retrieve the structured data.

ClarityNLP uses the `task_index` parameter to identify the single task that should execute the CQL query. Any NLPQL file can contain multiple invocations of `Clarity.CQLExecutionTask`. Each of these should have a `task_index` parameter, and they should be numbered sequentially starting with 0. In other words, each `define`

statement containing an invocation of `Clarity.CQLExecutionTask` should have a unique value for the zero-based `task_index`.

The `patient_id` parameter identifies the patient whose data will be accessed by the CQL query. This ID should match that specified in the documentset creation statement.

The remaining parameters from the table above are set to values appropriate for GA Tech's FHIR infrastructure.

The `cql` parameter is a triple-quoted string containing the CQL query. This CQL code is assumed to be syntactically correct and is passed to the FHIR server's CQL evaluation service unaltered. All CQL code should be checked for syntax errors and other problems prior to its use in an NLPQL file.

This example omits the optional time window parameters.

```
documentset PatientDocs:
 Clarity.createDocumentSet({
     "filter_query":"source:MYDOCS AND subject:99999"
 });

 define WBC:
     Clarity.CQLExecutionTask({
         documentset: [PatientDocs],
         "task_index": 0,
         "patient_id":"99999",
         "cql_eval_url":"https://gt-apps.hdap.gatech.edu/cql/evaluate",
         "fhir_data_service_uri":"https://apps.hdap.gatech.edu/gt-fhir/fhir/",
         "fhir_terminology_service_uri":"https://cts.nlm.nih.gov/fhir/",
         "fhir_terminology_service_endpoint":"Terminology Service Endpoint",
         "fhir_terminology_user_name":"username",
         "fhir_terminology_user_password":"password",
         cql: """
              library Retrieve2 version '1.0'

              using FHIR version '3.0.0'

              include FHIRHelpers version '3.0.0' called FHIRHelpers

              codesystem "LOINC": 'http://loinc.org'

              define "WBC": Concept {
                  Code '26464-8' from "LOINC",
                  Code '804-5' from "LOINC",
                  Code '6690-2' from "LOINC",
                  Code '49498-9' from "LOINC"
              }

              context Patient

              define "result":
                  [Observation: Code in "WBC"]
              """
     });

     context Patient;
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
documentset	*documentset*	Yes	Documents for a SINGLE patient only.
task_index	int	Yes	Each CQLExecutionTask statement must have a unique value of this index.
patient_id	str	Yes	CQL query executed on FHIR server for this patient.
cql_eval_url	str	Yes	See table above.
fhir_data_service_uri	str	Yes	See table above.
fhir_terminology_service_uri	str	Yes	See table above.
fhir_terminology_service_endpoint	str	Yes	See table above.
cql	triple-quoted str	Yes	Properly-formatted CQL query, sent verbatim to FHIR server.
fhir_terminology_user_name	str	No	Optional, depends on configuration of terminology server
fhir_terminology_user_password	str	No	Optional, depends on configuration of terminology server
time_start	str	No	Optional, discard results with timestamp < time_start
time_end	str	No	Optional, discard results with timestamp > time_end

Results

The specific fields returned by the CQL query are dependent on the type of FHIR resource that contains the data. ClarityNLP can decode these FHIR resource types: `Patient`, `Procedure`, `Condition`, and `Observation`. It can also decode bundles of these resource types.

Fields in the MongoDB result documents are prefixed with the type of FHIR resource from which they were taken except for the `datetime` field, which omits the prefix to enable date-based sorting. The prefixes for each are:

FHIR Resource Type	Prefix
Patient	patient
Procedure	procedure
Condition	condition
Observation	obs

The fields returned for the `Patient` resource are:

Field Name	Meaning
patient_subject	patient id
patient_fname_1	patient first name (could have multiple first names, numbered sequentially)
patient_lname_1	patient last name (could have multiple last names, numbered sequentially)
patient_gender	gender of the patient
patient_date_of_birth	date of birth in YYYY-MM-DD format

The fields returned for the `Procedure` resource are:

Field Name	Meaning
procedure_id_value	ID of the procedure
procedure_status	status indicator for the procedure
procedure_codesys_code_1	code for the procedure; multiple codes are numbered sequentially
procedure_codesys_system_1	code system; multiple code systems are numbered sequentially
procedure_codesys_display_1	code system procedure name; multiple names are numbered sequentially
procedure_subject_ref	typically the string 'Patient/' followed by a patient ID, i.e. Patient/99999
procedure_subject_display	patient full name string
procedure_context_ref	typically the string 'Encounter/' followed by a number, i.e. Encounter/31491
procedure_performed_date_time	timestamp of the procedure in YYYY-MM-DDTHH:mm:ss+hhmm format
datetime	identical to procedure_performed_date_time

The fields returned for the `Condition` resource are:

Field Name	Meaning
condition_id_value	ID of the condition
condition_category_code_1	category code value; multiple codes are numbered sequentially
condition_category_system_1	category code system; multiple code systems are numbered sequentially
condition_category_display_1	category name; multiple names are numbered sequentially
condition_codesys_code_1	code for the condition; multiple codes are numbered sequentially
condition_codesys_system_1	code system; multiple code systems are numbered sequentially
condition_codesys_display_1	code system condition name; multiple names are numbered sequentially
condition_subject_ref	typically the string 'Patient/' followed by a patient ID, i.e. Patient/99999
condition_subject_display	patient full name string
condition_context_ref	typically the string 'Encounter/' followed by a number, i.e. Encounter/31491
condition_onset_date_time	timestamp of condition onset in YYYY-MM-DDTHH:mm:ss+hhmm format
datetime	identical to condition_onset_date_time
condition_abatement_date_time	timestamp of condition abatement in YYYY-MM-DDTHH:mm:ss+hhmm format
end_datetime	identical to condition_abatement_date_time

The fields returned for the `Observation` resource are:

Field Name	Meaning
obs_codesys_code_1	code for the observation; multiple codes are numbered sequentially
obs_codesys_system_1	code system; multiple code systems are numbered sequentially
obs_codesys_display_1	code system observation name; multiple names are numbered sequentially
obs_subject_ref	typically the string 'Patient/' followed by a patient ID, i.e. Patient/99999
obs_subject_display	patient full name string
obs_context_ref	typically the string 'Encounter/' followed by a number, i.e. Encounter/31491
obs_value	numberic value of what was observed or measured
obs_unit	string identifying the units for the value observed
obs_unit_system	typically a URL with information on the units used
obs_unit_code	unit string with customary abbreviations
obs_effective_date_time	timestamp in YYYY-MM-DDTHH:mm:ss+hhmm format
datetime	identical to obs_effective_date_time

Collector

No

Clarity.GleasonScoreTask

Description

This is a custom task for extracting a patients Gleason score, which is relevant to prostate cancer diagnosis and staging.

Example

```
define final GleasonFinderFunction:
    Clarity.GleasonScoreTask({
        documentset: [Docs]
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

Name	Type	Notes
sentence	str	
start	int	
end	int	
value	int	Gleason score
value_first	int	First number in Gleason score
value_second	int	Second number in Gleason score

Collector

No

Clarity.PFTFinder

Description

Custom module for extracting pulmonary function test (PFT) values.

Examples

```
termset Terms:
  ["FEV1", "FEV", "PFT", "pulmonary function test"];

define final PFTTestPatients:
  Clarity.PFTFinder({
    termset:[Terms]
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	Yes	List of possible terms to find, e.g. 'NYHA'
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

Name	Type	Notes
sentence	str	Sentence where measurement is found
start	int	offset of the first character in the matching text
end	int	offset of the final character in the matching text plus 1
fev1_condition	str	
fev1_units	str	
fev1_value	floats	
fev1_text	str	
fev1_count	int	
fev1_fvc_ratio_count	int	
fev1_fvc_condition	str	
fev1_fvc_units	str	
fev1_fvc_value	float	
fev1_fvc_text	str	
fvc_count	int	
fvc_condition	str	
fvc_units	str	
fvc_value	float	
fvc_text	str	

Collector

No

Clarity.RaceFinderTask

Description

This is a custom task for extracting a patient's race (i.e. asian, african american, caucasian, etc.).

Example

```
define RaceFinderFunction:
    Clarity.RaceFinderTask({
        documentset: [DischargeSummaries]
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

Name	Type	Notes
sentence	str	
start	int	
end	int	
value	str	Race mentioned in note
value_normalized	str	Normalized value, e.g. caucasian -> white

Collector

No

Clarity.TextStats

Description

Task that uses textacy to get aggregate statistics about the text.

Example

```
Clarity.TextStats({
  documentset: [ProviderNotes]
});
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	
group_by	str	No	Default = *report_type*, the field that statistics be grouped on.

Results

Name	Type	Notes
avg_word_cnt	float	Average word count
avg_grade_level	float	Average Flesch Kincaid grade level
avg_sentences	float	Average number of sentences
avg_long_words	float	Average number of long words
avg_polysyllable_words	float	Average number of polysyllabic words

Collector

BaseCollector

Clarity.TNMStager

Description

Extract tumor stages from text. Read more *here*.

Example

```
define final TNMStage:
    Clarity.TNMStager ({
        cohort:PSAPatients,
        documentset: [Docs]
    });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

See the 'Outputs' table *here*.

Collector

No

Clarity.TransfusionNursingNotesParser

Description

Task that parses Nursing notes (specifically formatted for Columbia University Medical Center) for transfusion information.

Example

```
phenotype "TNN" version "2";

include ClarityCore version "1.0" called Clarity;

documentset TransfusionNotes:
    Clarity.createDocumentSet({
        "report_types":["Transfusion Flowsheet"]});

define TransfusionOutput:
    Clarity.TransfusionNursingNotesParser({
        documentset: [TransfusionNotes]
        });
```

Extends

BaseTask

Arguments

Name	Type	Required	Notes
termset	*termset*	No	
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

Name	Type	Notes
reaction	str	yes or no
elapsedMinutes	int	
transfusionStart	str	YYYY-MM-DD HH:MM:SS (ISO format)
transfusionEnd	str	YYYY-MM-DD HH:MM:SS (ISO format)
bloodProduc-tOrdered	str	
dateTime	str	YYYY-MM-DD HH:MM:SS (ISO format) at which these measurements were taken
timeDeltaMinutes	int	elapsed time in minutes since transfusionStart
dryWeightKg	float	
heightCm	int	
tempF	float	
tempC	float	
heartRate	int	units of beats/min
respRateMachine	int	units of breaths/min
respRatePatient	int	units of breaths/min
nibpSystolic	int	
nibpDiastolic	int	
nibpMean	int	
arterialSystolic	int	
arterialDiastolic	int	
arterialMean	int	
bloodGlucose	int	units of mg/dl
cvp	int	units mmHg
spO2	int	percentage
oxygenFlow	int	units of Lpm
endTidalCO2	int	units of mm Hg
fiO2	int	percentage

Collector

No

Base Classes

Also see the following classes, which are the base classes for the NLPQL tasks:

BaseTask

The base class for most ClarityNLP tasks. Provides most of the wiring needed to run individual algorithms.

Arguments

Name	Type	Required	Notes
termset	*termset*	See implementation	
documentset	*documentset*	No	
cohort	*cohort*	No	

Results

Name	Type	Notes
pipeline_type	str	Pipeline type internal to ClarityNLP.
pipeline_id	int	Pipeline ID internal to ClarityNLP.
job_id	int	Job ID
batch	int	Batch number of documents
owner	str	Job owner
nlpql_feature	str	Feature used in NLPQL *define*
inserted_date	date	Date result written to data store
concept_code	int	Code specified by user to assign to OMOP concept id.
phenotype_final	bool	*final* flag designated in NLPQL; displays in final results
report_id	str	Document report ID, if document level result
subject	str	Document subject/patient, if document level result
report_date	str	Document report date, if document level result
report_type	str	Document report type, if document level result
source	str	Document source, if document level result
solr_id	str	Document Solr *id* field, if document level result

Functions

run()

Main function that sets up documents and runs the task execution.

---

output()

Gets Luigi file, used for job communication or temp output.

---

set_name(name)

Sets name of task.

---

write_result_data(temp_file: File, mongo_client: MongoClient, doc: dict, data: dict, prefix: str='', phenotype_final: bool=False)

Writes results to MongoDB.

---

write_multiple_result_data(temp_file: File, mongo_client: MongoClient, doc: dict, data: list, prefix: str='')

Writes results to MongoDB as a list.

---

write_log_data(job_status: str, status_message: str)

Writes log message to the *job_status* table.

---

run_custom_task(temp_file: File, mongo_client: MongoClient)

The primary function for tasks to implement to run tasks.

---

get_document_text(doc: dict, clean=True)

Returns a string containing the text of a given Solr document.

---

get_boolean(key: str, default=False)

Looks up custom argument with matching key of type *bool*.

---

get_integer(key: str, default=-1)

Looks up custom argument with matching key of type *int*.

---

get_string(key: str, default='')

Looks up custom argument with matching key of type *str*.

---

get_document_sentences(doc)

Returns a collection of sentences for the given Solr document.

BaseCollector

The base class for ClarityNLP aggregate tasks. Only gets called after all the other tasks of its related type are complete.

Functions

run(pipeline_id, job, owner, pipeline_type, pipeline_config)

Main function that runs the collector.

---

run_custom_task(pipeline_id, job, owner, pipeline_type, pipeline_config, client, db)

Primary function where custom implementation of the collector is written.

---

custom_cleanup(pipeline_id, job, owner, pipeline_type, pipeline_config, client, db)

Run custom custom cleanup after collector has run.

---

cleanup(pipeline_id, job, owner, pipeline_type, pipeline_config)

Main cleanup task that marks job as complete and runs custom cleanup tasks after collector is completed.

1.5.3 NLPQL Operations

All operations are prefixed in NLPQL as *define*, with the optional *final* flag. The *final* flag writes each result as part of the finalized result set in MongoDB.

Data Operations

1.6 API Reference

1.6.1 NLP Web APIs

NLP endpoints provided by ClarityNLP.

/kill_job/<int:job_id>

GET pids of NLPQL tasks. Attemps to kill running Luigi workers. Will only work when NLP API and Luigi are deployed on the same instance.

/measurement_finder

POST JSON to extract measurements. Sample input JSON here.

/named_entity_recognition

POST JSON to run spaCy's NER. Sample input JSON here.

/nlpql

POST NLPQL plain text file to run phenotype against data in Solr. Returns links to view job status and results. Learn more about NLPQL *here* and see samples of NLPQL here.

/nlpql_tester

POST NLPQL text file to test if parses successfully. Either returns phenotype JSON or errors, if any. Learn more about NLPQL *here* and see samples of NLPQL here.

/nlpql_expander

POST to expand NLPQL termset macros. Read more *here*.

/nlpql_samples

GET a list of NLPQL samples.

/nlpql_text/<string:name>

GET NLPQL sample by name.

/phenotype

POST Phenotype JSON to run phenotype against data in Solr. Same as posting to */nlpql*, but with the finalized JSON structured instead of raw NLPQL. Using */nlpql* will be preferred for most users. See sample here.

/phenotype_feature_results/<int:job_id>/<string:feature>/<string:subject>

GET phenotype results for a given feature, job and patient/subject.

/phenotype_id/<int:phenotype_id>

GET a pipeline JSON based on the phenotype_id.

/phenotype_job_by_id/<string:id>

GET a phenotype jobs JSON by id.

/phenotype_jobs/<string:status_string>

GET a phenotype job list JSON based on the job status.

/phenotype_paged_results/<int:job_id>/<string:phenotype_final_str>

GET paged phenotype results.

/phenotype_result_by_id/<string:id>

GET phenotype result for a given mongo identifier.

/phenotype_results_by_id/<string:ids>

GET phenotype results for a comma-separated list of ids.

/phenotype_structure/<int:id>

GET phenotype structure parsed out.

/phenotype_subject_results/<int:job_id>/<string:phenotype_final_str>/<string:subject>

GET phenotype results for a given subject.

/phenotype_subjects/<int:job_id>/<string:phenotype_final_str>

GET phenotype_subjects.

/pipeline

POST a pipeline job (JSON) to run on the Luigi pipeline. Most users will use */nlpql*. Read more about pipelines here. See sample JSON here.

/pipeline_id/<int:pipeline_id>

GET a pipeline JSON based on the pipeline_id.

/pipeline_types

GET a list of valid pipeline types.

/pos_tagger

POST JSON to run spaCy's POS Tagger. (Only recommended on smaller text documents.) Sample input JSON here.

/report_type_mappings

GET a dictionary of report type mappings.

/sections

GET source file for sections and synonyms.

/status/<int:job_id>

GET status for a given job.

/term_finder

POST JSON to extract terms, context, negex, sections from text. Sample input JSON here.

/tnm_stage

POST JSON to extract TNM staging from text. Sample input JSON here.

/value_extractor

POST JSON to extract values such as BP, LVEF, Vital Signs etc. Sample input JSON here.

/vocabExpansion?type=<TYPE>&concept=<CONCEPT>&vocab=<VOCAB>

About:

This API is responsible for vocabulary explosion for a given concept. API accepts a _type_ which can be synonyms, ancestors or descendants. The API has to accept the _concept_ name which is supposed be exploded. The vocabulary _vocab_ can also be passed as an optional parameter.

Parameters:

- Type: mandatory - 1: synonyms - 2: ancestors - 3: descendants
- Concept: mandatory

- Vocab: optional

Example usage:

‘‘‘ http://nlp-api:5000/vocabExpansion?type=1&concept=Inactive

http://nlp-api:5000/vocabExpansion?type=1&concept=Inactive&vocab=SNOMED ‘‘‘

/ngram_cohort

GET

About:

Generating n-grams of the Report Text for a particular Cohort. API has to accept the Cohort ID, the _n_ in n-gram, and frequency (the minimum occurrence of a particular n-gram). The API also accepts a keyword. If given the keyword, only n-grams which contain that keyword are returned.

Parameters:

- Cohort ID : mandatory
- Keyword : optional
- n : mandatory
- frequency : mandatory

Example usage:

```
~/ngram_cohort?cohort_id=6&n=15&frequency=10

~/ngram_cohort?cohort_id=6&keyword=cancer&n=15&frequency=10
```

OHDSI WebAPI Utilities

/ohdsi_create_cohort?file=<FILENAME>

- Description: - Creating cohorts using OHDSI web API. - API requires a JSON file which contains cohort creation details. - JSON file must be placed in */ohdsi/data/* - *test_cohort.json* is an example file which depicts the JSON structure which needs to be *strictly* followed.
- Method: GET
- Parameters: - JSON file name
- Usage: ` `http://nlp-api:5000/ohdsi_create_cohort?file=<FILENAME>` `

/ohdsi_get_cohort?cohort_id=<COHORT_ID>

- Description: Get cohort details from OHDSI.
- Method: GET
- Parameters: - cohort_id
- Usage: ` `http://nlp-api:5000/ohdsi_get_cohort?cohort_id=<COHORT_ID>` `

/ohdsi_get_cohort_by_name?cohort_name=<COHORT_NAME>

- Description: Get Cohort details by name
- Method: GET
- Parameters: - cohort_name
- Usage: ` http://nlp-api:5000/ohdsi_get_cohort_by_name?cohort_name=<COHORT_NAME> `

/ohdsi_get_conceptset?file=<FILENAME>

- Description: - Getting concept set info using OHDSI web API. - API requires a JSON file which contains concept set details. - JSON file must be placed in */ohdsi/data/* - *test_concept.json* is an example file which depicts the JSON structure which needs to be *strictly* followed.
- Method: GET
- Parameters: - JSON file name
- Usage: ` http://nlp-api:5000/ohdsi_get_conceptset?file=<FILENAME> `

/ohdsi_cohort_status?cohort_id=<COHORT_ID>

- Description: Get the status of the triggered cohort creation job.
- Method: GET
- Parameters: - cohort_id
- Usage: ` http://nlp-api:5000/ohdsi_cohort_status?cohort_id=<COHORT_ID> `

1.7 Frequently Asked Questions (FAQ)

1. **How can I check the syntax of my NLPQL file without actually running it?**

   Send your NLPQL file via HTTP POST to the *nlpql_tester* API endpoint. ClarityNLP will return a a JSON representation of your file if the syntax is correct. If a syntax error is present, ClarityNLP will print an error message and no JSON will be returned.

2. **If I'm using the Docker version of ClarityNLP, how do I verify that all the supporting Docker containers are up and running?**

   Open a terminal and run the command `docker ps`. The status of each container will be printed to stdout. Each container should report a status message of `Up n seconds`, where n is an integer, if the container is fully initialized and running.

1.8 Troubleshooting Guide

1.9 About

1.9.1 Team

The team works from the Information and Communications Laboratory (ICL) at the Georgia Tech Research Institute (GTRI).

The team is closely associated with the Georgia Tech Center for Health Analytics and Informatics (CHAI).

Members

- Jon Duke
- Charity Hilton
- Richard Boyd
- Trey Schneider
- Christine Herlihy
- Caleb Sides

Current Students

Former Students

- Chirag Jamadagni
- Prathamesh Prabhudesai

1.9.2 Partners

We're currently collaborating with Celgene and the FDA.

1.9.3 Projects

CHAPTER 2

Contact Us

Feel free to connect with us on Slack.

CHAPTER 3

License

This project is licensed under Mozilla Public License 2.0.