DENSE FORECASTING OF WILDFIRE SMOKE PARTICULATE MATTER USING SPARSITY INVARIANT CONVOLUTIONAL NEURAL NETWORKS

Sparse Spatiotemporal Prediction of Wildfire Smoke Particulate Matter Via Convolutional Neural Networks

Report Prepared By

Ashutosh Bhudia
Brandon Dos Remedios
Minnie Teng
Ren Wang

University of British Columbia
Vancouver

2020
UBC Data Science for Social Good
BC Centre for Disease Control

1 Abstract

Accurate forecasts of fine particulate matter (PM2.5) from wildfire smoke are crucial to safeguarding cardiopulmonary public health. Existing forecasting systems are trained on sparse and inaccurate ground truths, and do not take sufficient advantage of important spatial inductive biases. In this work, we present a convolutional neural network which preserves sparsity invariance throughout, and leverages multitask learning to perform dense forecasts of PM2.5 values. We demonstrate that our model outperforms two existing smoke forecasting systems during the 2018 and 2019 wildfire season in British Columbia, Canada, predicting PM2.5 at a grid resolution of 10 km, 24 hours in advance with high fidelity. Most interestingly, our model also generalizes to meaningful smoke dispersion patterns despite training with irregularly distributed ground truth PM2.5 values available in only < 0.5% of grid cells.

Contents

1 Abstract **1**

2 Introduction **3**
2.1 Project Motivation . . . 3
2.2 Project Strategy . . . 3
2.3 Data Science for Social Good . . . 3

3 Materials and Methods **4**
3.1 Datasets . . . 4

4 CNN Architecture **6**
4.1 Sparsity Invariant CNNs . . . 6
4.2 Multitask Learning . . . 6

5 Results and Discussion **7**

6 Conclusion **8**

Appendices **10**

A Data-Download-Suite **10**
A.1 Overview . . . 10
A.2 smoke/utils package . . . 10
A.3 smoke/ package . . . 12

B Data-Processing-Pipeline **14**
B.1 Overview . . . 14
B.2 Pre-cleaning steps . . . 16
B.3 Parsing Step . . . 18
B.4 Cleaning Step . . . 22
B.5 NOAA Processing . . . 32
B.6 BC MoE $PM_{2.5}$ Label Processing . . . 35
B.7 Grid Representation . . . 35
B.8 Amalgamator . . . 37
B.9 Validation . . . 40
B.10 Recommendations . . . 41

2 Introduction

2.1 Project Motivation

Wildfire smoke leads to episodes of poor air quality in British Columbia (BC), and exposure has been associated with a wide range of acute health effects from respiratory symptoms through to premature mortality. Furthermore, recent research indicates that these health effects can be experienced within hours of smoke exposure, highlighting the absolute importance of reliable wildfire smoke forecasts for protecting public health. However, forecasting wildfire smoke is even more challenging than forecasting the weather given the many uncertainties in fire location, fire growth, fuel consumption, emissions factors, and plume rise. As such, there is growing agreement that ensemble smoke forecasts may be more useful than any single smoke forecast in isolation.

2.2 Project Strategy

Air quality impacts of wildfires are currently communicated using a combination of air pollution measurements at sparsely distributed monitoring networks, simulated forecasts of air quality, and satellite imagery. Artificial intelligence (AI) algorithms can process vast quantities of data; its ability to process information from multiple sources efficiently may help to improve wildfire smoke forecasts. Here, we present an example of a convolutional neural network, specifically the UNet, which we propose to be capable of predict presence and magnitude fire smoke presence using input data sources from simulated forecasts and satellite imagery. We aim to demonstrate this concept by predicting retrospective wildfire smokes in BC using data from 2018-2019.

2.3 Data Science for Social Good

Early warnings about smoke are critical for protecting population health, and better predictions lead to better public health outcomes. This model attempts to improve on current wildfire smoke models and the improvements may potentially be used by fire rescue services, health and environmental agencies and the general public to better manage the health risks for fire smoke. With this model we are using PM2.5 values collected from approximately 80 air monitoring stations as ground truths to compare our predictions to. However, the 80 stations are disproportionately scattered throughout the province, with stations mostly concentrated in more populated areas.

Basing ground truths on disproportionately located stations may introduce bias to the model, and potentially leading to less accurate or available predictions for Indigenous Canadians that live in rural or remote areas.

3 Materials and Methods

3.1 Datasets

3.1.1 Forecasting Models

Various deterministic and statistical models for predicting PM2.5 exist. Inspired by paradigms in residual learning, we incorporate these baseline forecasting models as inputs to our convolutional model. The idea is to allow our network to leverage prior knowledge contained within other models, such as influences of meteorological conditions, fire behavior evolution and smoke dispersion mechanics. We then simply learn a function which models potential improvements to these baselines in order to better attain the ground truth PM2.5 values. In particular, we incorporate two prominent smoke particulate matter forecasting systems, FireWork and BlueSky Canada over the grid defined in section 3.1.5.

3.1.2 Meteorological Data

Previous work has shown that the Aerosol Optical Depth (AOD) metric is a meaningful proxy for PM2.5. We therefore include AOD from the U.S. National Aeronautics and Space Administration (NASA) Moderate Resolution Imaging Spectroradiometer (MODIS) instruments, available every 6 hours in a 24 hour period. We also include meteorological information from the NASA Modern Era Retrospective Analysis for Research and Applications, Version 2 (MEERA-2) program. We use eastward and northward components of wind vectors 50m above the surface, and at the 250 hPa and 500 hPa pressure levels, with a spatial resolution of $0.5^\circ \times 0.625^\circ$ (latitude $\times$ longitude).

3.1.3 Wildfire Data

MODIS also provides data on fire locations and intensity. Intensities are approximated by the fire radiative power (FRP) variable, and fire locations are specified by a weighted centroid localization of FRP values in all 1-km $\times$ 1-km fire pixels (as determined by the active fire product). Additionally, we include direct observations of smoke plumes from wildfires from the U.S.

National Oceanic and Atmospheric Administration (NOAA. Analysts hand draw these smoke plume boundaries based on available data from various fire detection sources.

3.1.4 Particulate Matter Ground Truth

For ground truth PM2.5 values, we use the 2018 and 2019 1-hour average PM2.5 measurements from 56 air quality monitoring stations through British Columbia (courtesy of the British Columbia Ministry of Environment and Climate Change Strategy). We log transform the values as per to address its heavy right-skewed distribution.

3.1.5 Multidimensional Wildfire Composition Images

In order to amalgamate these varied data sources into a temporally and spatially consistent format which can be consumed by our convolutional neural network we first define a regular, approximately square grid over the province of British Columbia, Canada, with (latitude, longitude) corners at (57.87, -133.54), (47.31, -127.18), (60.61, -112.19), (49.43, -110.61), and a grid resolution of 10-km x 10-km cells covering the roughly 1250-km x 1250-km area. This will serve as an image-like canvas on which we can populate different pixels with the requisite features.

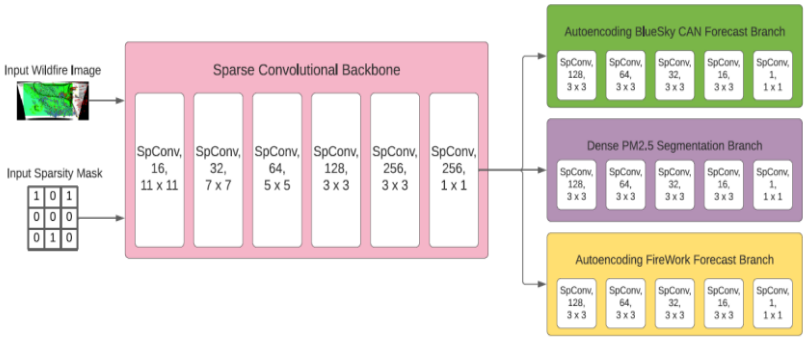


Figure 1: Multiheaded model architecture. Each SpConv layer involves a sparse convolution layer as described in with the indicated kernel size and number of filters, with average pooling of the sparsity mask, followed by a non-linear activation (ReLU throughout, except the final layer in the individual task branches, where no activation is used).

Second, note that we seek to make predictions 24-hours in advance. We

extract temporal and spatial (latitude/longitude) labels for each element of each dataset. Then, for each time where we have PM2.5 ground truths, we project available datapoints from all datasets 24-hours prior to their corresponding cells and channels within our defined grid. Cells and channels for unavailable measurements are simply set to the closest available measurements from an earlier time, or to -1 when even those are absent. We also do a similar projection for the available PM2.5 ground truths.

The final result is a $125 \times 125 \times 9$-dimensional input image for each timepoint of predictive interest, and a corresponding sparse $125 \times 125 \times 1$-dimensional output image of PM2.5 labels corresponding to 24-hours thereafter. In total, we train on 4870, validate on 610 and test on 610 such input-output pairs, randomly shuffled.

4 CNN Architecture

4.1 Sparsity Invariant CNNs

Sparsity invariant CNNs have been used as a means to preserve sparsity constraints throughout all layers of a convolutional neural network by explicitly accounting for a binary mask describing the sparsity pattern. In particular, such a "sparse" convolution involves pooling only over observed pixels of the image, and normalizing according to the mask. Here, our sparsity mask describes the locations of the available ground truth PM2.5 values over the spatial grid. While [11] use max pooling to downsample the binary mask after each sparse operation, we find that average pooling leads to smoother inpainting of the resultant PM2.5 output map. We employ these sparsity invariant layers in a core feature extraction backbone for FIRENET, as seen in Fig. 1.

4.2 Multitask Learning

Inspired by multitask learning to reduce overfitting, we also define additional autoencoding tasks (see top and bottom branches in Fig. 1 which each output a $125 \times 125 \times 1$ map, consistent with the original PM2.5 forecasts of the FireWork and BlueSky Canada models). We hope to provide learning signal where ground truths are unavailable, allowing FIRENET to overcome extreme sparsity issues by borrowing from the learned dynamics contained within the baseline forecasting models. Then our final model loss L is defined as:

$$\mathcal{L} = \gamma_1 ||I_{fw} - \hat{I}_{fw}||_1 + \gamma_2 ||I_{bscan} - \hat{I}_{bscan}||_1 + \gamma_3 ||I_{pm25} - M \odot \hat{I}_{pm25}||_1$$

where $\gamma_{1,2,3}$ are hyperparameters, I is the target map, $\hat{I}$ is the predicted map from the corresponding branch, with subscripts fw, $bscan$, $pm25$ denoting the FireWork baseline, BlueSky Canada baseline and $PM_{2.5}$ ground truths, respectively. M denotes the binary mask demarcating the sparsity pattern of the $PM_{2.5}$ ground truths.

Table 1: We report average L1 error for our model and available baselines for early, mid and late temporal subsets of the test set. Lower is better, bolded is best.

	Mean Absolute Error ($\mu g/m^3$)		
	Early (Apr + May)	**Mid** (Jun + Jul + Aug)	**Late** (Sep + Oct)
FIREWORK	10.08	23.39	17.52
BLUESKY CANADA	22.34	75.73	44.10
FIRENET	**2.26**	**14.21**	**6.79**

5 Results and Discussion

We assess performance of FIRENET by comparing model PM2.5 predictions with FireWork and BlueSky Canada PM2.5 predictions at the 56 air monitoring stations over timepoints within the defined test set. We also look at heatmaps of model predictions to ascertain whether or not meaningful interpolations are made in regions where ground truths are not available.

Table 1 details our model performance against the described baselines. Because PM2.5 values can be dramatically higher during the peak of the fire season in July and August, we separate this assessment for the early, mid and late fire seasons (April to May, June to August, and September to October, respectively). FIRENET outperforms both FireWork and BlueSky Canada at all points in the fire season, validating our residual learning approach and verifying that additional raw data is semantically useful for our model in improving the baseline predictions.

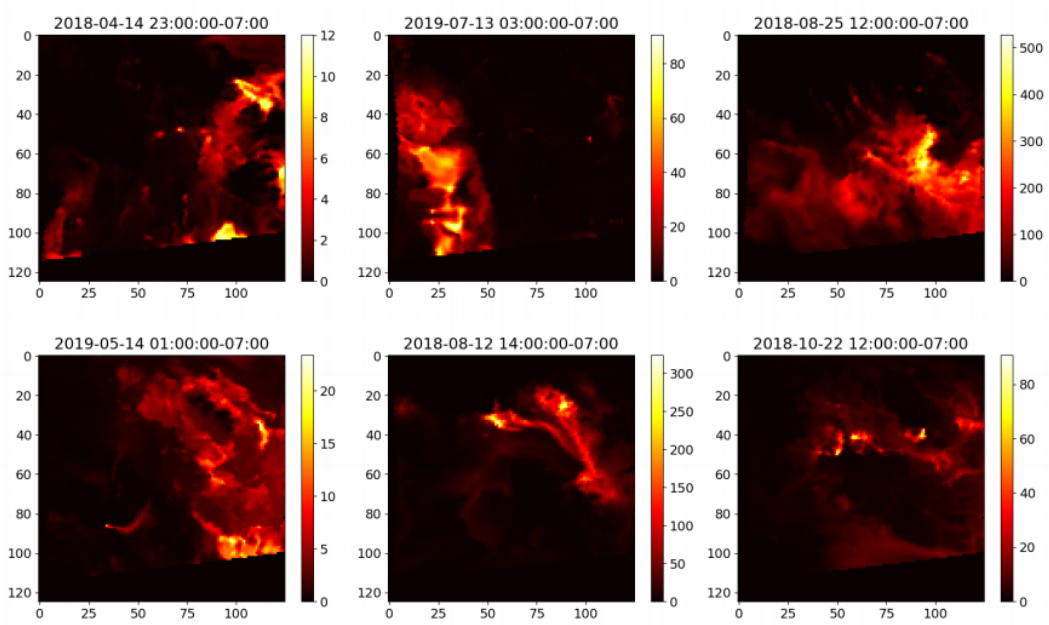


Figure 2: Predictions made using available data 24 hours in advance. We note the smooth inpainting of $PM_{2.5}$ values despite lacking ground truths on nearly 99.7% of the image.

In Fig. 2, we show model predictions over the entire defined grid. Each pixel within each heatmap represents a 100 km2 area. Firstly, note that we accurately capture highs and lows of PM2.5 in correspondence with the fire season (beginning in April, peaking in July and August, and ending in October). Secondly, we see that despite lacking ground truth PM2.5 values in between air monitoring stations, FIRENET is able to interpolate PM2.5 meaningfully, representing complex and diverse PM2.5 falloff patterns and interactions between smoke dispersion from various fires. More work clearly needs to be done to verify these implied dynamics, and we leave this for the future.

6 Conclusion

In this work, we tackle the challenging but important task of forecasting the public health burden of smoke particulate matter perpetuated by wildfires. By incorporating baseline forecasting models and raw meteorological and wildfire data variables from satellite measuring systems, we are able to design an input data format which preserves spatial relationships. We then overcome sparsity issues plaguing more traditional statistical modeling frameworks by introducing sparsity invariant layers, and defining auxiliary

tasks that provide guiding intermediate learning signal to the network. We demonstrate strong results on real world wildfires as compared to forecasting systems currently in use. Future work will ascertain that our method is generalizable outside British Columbia, Canada, and that the smoke particulate matter behaviors modeled by our network are consistent with domain expectations.

Appendices

A Data-Download-Suite

A.1 Overview

This package is used to download model input (forcing) files. This is necessary because different download providers have different requirements, thus automation is required. It also provides APIs (application Programming Interfaces) for developing downloaders that can fetch files from HTTP and FTP servers, and from NASA's LAADS download portal through scripting. That is, it provides functionality including web scraping, authentication, multi-threaded downloads and so on.

A.1.1 Code Structure

smoke/utils contains the core code providing APIs for scripting

smoke/ contains higher level code, which is used to do the actual scripting for the downloading, as well as provide user interface(s). An exemplar configuration file, and a proposed means of implementing a download deamon are also included.

A.2 smoke/utils package

A.2.1 conf_tools.py

Provides tools for validating and loading a download configuration file. It is implemented as a python function wrapper, and can simply be added as a decorator to any function that consumes a smoke_forcing configuration file.

It ensures that:

- The function contains the required 'config' argument, and that it is provided by the user
- The configuration file exists and is a valid file
- The configuration is passed on to the function as a python dictionary

A.2.2 download.py

This module contains the base class ('Download') that provides the methods that all other download processes must contain. The class itself provides the APIs for HTTP(S) downloads.

Here is a description of the methods, and what they do:

- ‘mkdir‘ checks that the directory on the local computer to which we want to download files and (1) checks if is accessible and (2) we have the rights to write to it (3) creates the directory if it does not already exist
- ‘get_checksum‘ is a static method that is capable of consuming files and returning its cksum, md5, sha1, sha512 or crc32 checksum (as specified)
- ‘download‘ fetches files and saves them to the desired location. A maximum of three retries are made to account for, say, network issues. The checksum is also verified with the server if required
- ‘get_file_name‘ is used to obtain the file name and extension from the download URL
- download_concurrently uses a thread pool to download files concurrently. It is generalised, and requires only an override of the ‘download‘ method to adapt its behavior to inheriting classes

A.2.3 ftp.py

This module extends the ‘Download‘ class, and is used to authenticate with and download files from an FTP server.

Here is a description of the methods, and what they do:

- ‘reset_connection is used to authenticate with FTP servers, and raises an error in the event of failure
- ‘download‘ overrides the original method, and is tailored for the FTP protocol
- ‘make_download_dict‘ is used to recursively traverse and FTP server’s folders to look for files, and creates a nested python dictionary of file locations in preparation for the download process

A.2.4 url_downloader.py

Extends the ‘Download‘ class. Overrides the ‘download‘ method to enable chunked downloads of large files for memory efficiency.

A.2.5 url_parse.py

Provides generic high level functions for web scraping.

- ‘get_auth‘ returns an authentication token
- ‘parse_protected_html‘ produces a BeautifulSoup parsed webpage
- ‘get_url_list‘ return a list of links from a parsed webpage
- ‘filter_by_ext‘ enables filtering of URLs by file extension

A.3 smoke/ package

The root of the package includes higher level scripting that uses the APIs from the ‘smoke/utils‘ package. The basic hierarchy is:

1. ‘urls.py‘ contains classes used to fetch/build and compile a list of URLs from each download source
2. ‘downloaders.py’ contains classes used to download files from each download source, provided the URLs
3. ‘main.py‘ provides the user interface and runs the downloaders

The instances that have been implemented are heavily documented, and it is recommended that they be used as templates for future additions.

A.3.1 urls.py

urls.py has a ‘URLs‘ abstract base class with methods:

- ‘generate_dates‘, a static method, which can be used to return a list of dates within a given range at a given interval
- ‘get_urls‘, the abstract static method that each inheriting class must implement. It is the main method called to generate URLs
- ‘validate_url‘ uses a maximum of 3 retries to communicate with an HTTP server to verify whether a URL is valid. This is particularly useful when URLs are generated from a pattern and not scraped from a webpage or server
- ‘validate_urls‘ uses a thread pool to validate URLs concurrently
- ‘group by year‘ is used to return a dictionary of lists of URLs sorted by year.

A.3.2 downloaders.py

downloaders.py has a ‘Downloader‘ abstract base class with methods:

- ‘run‘, an abstract method. It is the main method to be executed by the ‘main‘ program. essentially, when implemented, it uses the list of URLs, sets up the local directories for downloading, and monitors the download process.
- ‘parse_time‘, a static method that can be used to parse strings into a python datetime

A.3.3 config

An exemplar configuration file is included in the repository. It is recommended to use a separate git repository to track the configuration file. Hashtags can be used to comment out block directives, in which case the program will automatically ignore them. The directives themselves can be listed in any particular order, provided only that the hierarchy is maintained.

A.3.4 Installation

To install the package, these are the steps to be followed (in a linux terminal. Ensure you have python 3.8 and git installed in the terminal):

1. Clone the repository. Open a terminal, and navigate to the directory where you wich to clone the repository. Then execute ‘git clone git@gitlab.math.ubc.ca:smoke/smoke_forcing.git‘
2. ‘cd‘ into the cloned repository, and execute ‘pip install –user –editable‘. this will install the python package on your interpreter, and allow it to be modified if chnages ar emade to the repository.
3. To your ‘ .bashrc‘ file, add the following ‘export PATH= /.local/bin:$PATH‘ and save the file. Then execute ‘source /.bashrc‘ to refresh.

A.3.5 Usage

Due to the installation, the command-line tool is now available system wide. After you have edited your ‘config.yml‘ file as desired, save it and:

1. Open a terminal and execute ‘smoke-forcing $path-to-config‘. That’s it!

Use ‘smoke-forcing –help‘ to bring up the help dialog in the terminal. The debugging options can be used to provide detailed dumps of execution progress and errors. If a download is anticipated to take a considerable amount of time, especially over SSH where there is the risk of being logged out, consider using the ‘tmux‘ utility on the computer where the files are being downloaded. This has the added benefit of being able to check in on the download progress from any other machine (under the same user account).

B Data-Processing-Pipeline

B.1 Overview

B.1.1 Description

The data processing pipeline is the code base that operates between the download suite of smoke_forcing and the model of smoke-models. This pipeline has the primary function of converting raw data files downloaded from the download suite of smoke_forcing, to prediction data tensors that the CNN model can train off of. For the Firework dataset, the BlueSky dataset, the MODIS AOD dataset, and with minor alterations the MODIS FRP dataset, this conversion is done through a three step process, of parsing raw data files, cleaning/converting data into saved grids, then amalgamating data together into a tensor. NOAA and BC MoE data require different loading steps which will be explained in sections B.5 and B.6 respectively.

B.1.2 Code Structure

The code base of the data processing pipeline is separated into two major directories after the root directory of smoke, which are smoke/smoke and smoke/tests. The smoke/smoke directory contains all of the pipeline infrastructure for the conversion process, and will be the main directory to look in for the code that is being described in the B Data-Processing-Pipeline section. The smoke/smoke directory is divided further into 7 subdirectories named amalgamate, box, clean, convert, load, noaa, and split. smoke/smoke/amalgamate contains the Amalgamator and run_amalgamator script which are used for creating the final pytorch tensors from saved grids, explained in B.8. smoke/smoke/box contains the Box, FeatureTimeSpaceGrid, and TemporaryTimeSpaceGrid which describe the grid and store data regarding it, explained respectively in B.7.1, B.7.3, and B.7.4. smoke/smoke/convert contains scripts for saving out NetCDF4 files which result from the parsing steps of B.3 if those types of files are desired. smoke/smoke/clean

contains the cleaner tool set of the CellCruncher and TimeCruncher, the run_cleaners script, and all cleaners sans the noaa cleaner, which are all used in the process of converting parsed data to grids, and are all explained throughout B.4. smoke/smoke/load contains the GeographicalDataset, and all parsers sans the noaa parser, which are all used in the process of parsing raw data files, explained throughout B.3. smoke/smoke/noaa contains the Geohashparser and BoxMapper objects which in tandem cover the NOAA parsing and cleaning process from raw data file to grid, and are explained throughout B.5. Finally, smoke/smoke/split contains the functions for splitting the MODIS FRP fire archive shape file, and is explained in B.2.1. The smoke/tests directory contains sub-directories corresponding to major ones within smoke/smoke which test the objects within the directories of their namesake. These directories mainly contain unit testing for the objects with an interesting exception on the manual visual testing of the cleaners in smoke/tests/clean, and will be described in B.9 Validation.

B.1.3 Full Ordered Steps

This subsection will describe the ordered steps necessary to use the pipeline to convert raw data files into pytorch tensors in the standard fashion which was done for the data in this report. The following legend will apply for these steps, stating which datasets require any one particular step, and thus that the particular step must be performed in order to place that dataset's data in the resulting pytorch tensor:

- FW - Firework
- BS - BlueSky Canada
- MA - MODIS AOD
- MF - MODIS FRP
- NO - NOAA Smoke Plume

Note: This process assumes that raw data files using the section A Download-Suite have already been downloaded, and have known locations on the system.

1. (MF) Split the fire_archive shape file into individual hdf files into some output directory, using either the smoke/smoke/frp_splitter.py script or by importing partition_to_hour from that .py file, as explained in B.2.1.

2. (FW, BS, MA, MF) Adjust the smoke/smoke/clean/run_cleaners_config.yml file (or some other yaml file with a similar format) to specify grid resolution in km, time range to clean/convert a saved FeatureTimeSpaceGrid grid daily inside, parameters of which datasets you want to clean (by adjusting the run

parameter), and various parameters specific to each dataset pertaining to raw data file location, data window size, etc. This is explained more in B.4.6.

3. (FW, BS, MA, MF) Run the smoke/smoke/clean/run_cleaners.py script giving the path to the previous adjusted yaml file. This step will load information from the raw data files, process them into consistent grids, and save those grids into FeatureTimeSpaceGrid's tar.gz files. These processed files are simply objects containing data for the dataset at a location time and space, for a particular data measurement ("feature") such as $PM_{2.5}$ forecast or corrected optical depth at 0.47 microns, for example. This object is explained more in B.7.3.

4. (NO) Run the NOAA processing branch as per instructions in Minnie's video and throughout B.5.

5. (FW, BS, MA, MF, NO) Adjust the smoke/smoke/amalgamate/run_amalgamator_config.yml file (or some other yaml file with a similar format) to specify time range for tensors, time resolution (e.g. one tensor for every hour), paths to directories containing the saved processed files, and which datasets to include in the tensor (if the dataset is not desired there is no need to clean it).

6. (FW, BS, MA, MF, NO) Run the smoke/smoke/amalgamate/run_amalgamator.py script, giving the path to the previously adjusted config, and an output directory for the resulting tensors. This will parse the config then, using the specifications, will generate prediction data pytorch tensors for every specified time that has a complete prediction set (valid prediction data at that point in time across all datasets added), which is explained further in B.8. This will complete the pipeline process leaving one with a directory full of prediction data tensors.

B.2 Pre-cleaning steps

B.2.1 FRP Shape File Splitting

This subsection will explain the MODIS FRP shape file splitting process, explaining why it was made and how to use it to split an FRP fire archive shape file. This splitting process takes place using the smoke/smoke/split/frp_splitter.py file with either the partition_to_hour function contained within, or by running the .py file as a whole. This splitting process is done in order to get the information contained within the MODIS

FRP fire archive .shp file, into the form of individual files with smaller periods of time and with data held within a proper grid of latitudes and longitudes instead of just points This such that the FRP data is similar to that of Firework, BlueSky Canada, and MODIS AOD files. Doing this will allow the MODIS FRP dataset to use the abstracted GenericParser, GenericCleaner, and other associated objects, which were made for files like Firework, Bluesky, or MODIS AOD, giving it full access to that entire code suite which makes code maintenance and development much easier. This also prevents us from having to load the entire FRP shp file when generating our grids, which saves on time and prevents us from overloading RAM. The drawback would of course be that this adds an additional step and takes up additional drive space, but the gains in centralized code, speed, not having to recreate a ton of code, and other benefits far outweigh this.

The FRP splitting process is merely an explanation of what happens in the partition_to_hour function as any splitting, including the main script run of the .py file, simply involve the calling of the function. This splitting process is as follows, and assumes that the shape file is given with all necessary components to load:

1. Load the shape file using GeoPandas and convert it into a xarray Dataset which will contain rows of individual FRP measurements with columns of ACQ_DATE/ACQ_TIME (when point was measured), FRP (the frp value), and LATITUDE/LONGITUDE (where point was measured)

2. Extract from the rows, all ACQ_DATE/ACQ_TIME to get a full list of individual times present in the dataset, then pull from all of these the unique set of datetimes to the hour. This will allow us to group all data to files based on which unique hour they were measured within. Grouping based on hour was chosen as grouping on each day created individual files that were multiple gigabytes in size which is too large, and grouping on minute created far too many individual files.

3. For each unique hour in the data set, the following process will be performed

    - Gather all FRP points that occur during that hour and extract out their times, positions, and frp measurements
    - Get a unique list of latitude and longitude for all points, used as the 2 space axes, and form a grid with the meshing of both lists
    - For each time make a grid of the size defined by the space axes

- Populate the grid with data at that time, at its corresponding latitude and longitude position in the grid, leaving the rest as nan values
- Place the populated grid of data in an xarray DataArray and Dataset with the ordered time axis, latitude axis, and longitude axis as coordinates
- Save out the xarray Dataset into an hdf of data in that hour

After that process is completed for each hour, the shape file will have been completely partitioned in hdf files each representative of one hour in the MODIS FRP dataset.

The splitter can be used either by importing and calling the partition_to_hour function from smoke.clean.split.frp_splitter.py or by running the frp_splitter.py file from the command line. In either case, the two arguments necessary to specify are the path to the shape file to split, and an output directory to store the split hdf files. The following is an example call to frp_splitter.py from the command line, assuming one's working directory is smoke/smoke/split

```
python frp_splitter.py <frp.shp file path> <output directory path>
```

B.3 Parsing Step

B.3.1 GeographicalDataset

The GeographicalDataset object is contained within smoke/smoke/datasets.py. It was created to serve as a common data object, such that accessing geographical data for all differing datasets and data files, could be done in the exact same way after being loaded into one of these objects. This object contains information about feature (what is being measured), time, latitude, and longitude for each data measurement in the data file that is loaded into it.

A GeographicalDataset object is essentially a wrapper around an xarray Dataset with standardized coordinate names and data storage form. The inner Dataset of any GeographicalDataset, is data is stored in 3 dimensional grids of axes time, latitude, and longitude in that order, with one 3D grid for each feature contained within. To extract information from a GeographicalDataset one can call any of get_latitudes, get_longitudes, get_features, and get_times to pull out an array of the ordered entries along that particular

coordinate axis. To pull out one of the 3D grids for a feature, one can use the get_feature_data_array method, with an argument of the feature name (which can be found from the get_features method), which will return an xarray DataArray of gridded data across time, latitude and longitude.

The requirements to create a GeographicalDataset are essentially an xarray Dataset containing an arbitrary amount of features each with a corresponding DataArray. Each of these DataArrays must share an identical 3 dimensional coordinate system labelled appropriately with the names "time" for time, "lat" for latitude, and "lon" for longitude. Data can then populate the DataArray as appropriate. Creating a GeographicalDataset requires passing one xarray Dataset of these specifications as an argument, and GeographicalDataset will check to make sure that the requirements are met.

B.3.2 GenericParser

The GenericParser is an abstract object that the other parsers extend, which aims to hold functions to convert a raw data file at some given path into a standardized GeographicalDataset object. It does this with the parse_file method which calls an abstract method convert_raw_to_dataset on a raw data file path given, using the abstract method to load and manipulate the data of the file into an xarray Dataset of the specifications for a GeographicalDataset, then creates a GeographicalDataset with that xarray Dataset, and returns the GeographicalDataset. All extended objects (all parsers) implement this convert_raw_to_dataset method, and thus contain the process necessary to convert files of that specific dataset, to the form of a GeographicalDataset.The GenericParser object is located in smoke/smoke/load/parsers.py along with all of the other parsers from B.3.3 onwards.

B.3.3 FireworkParser

The FireworkParser object is an implementation of the GenericParser object which overrides the convert_raw_to_dataset method with one containing the process for specifically converting a Firework Geotiff file into a GeographicalDataset object. This process is as follows:

1. Load the FireworkGeotiff, via a given file path, with xarray's open_rasterio function, which uses rasterio as a backend, and loads the geotiff into a standard xarray DataArray object of rows of individual data points, with columns of x, y, and band

2. Rename the x column to “lon”, y column to “lat”, and band to “time”, which are the proper naming specifications required by the GeographicalDataset object, and are true to what x, y, and band represent in the geotiff

3. Since start time is not included in meta data of the Firework Geotiff, extract the starting time from the file name

4. Since each “band” (now labelled “time”) represents n band hours in the future from the zeroth hour forecast (current state) of the zeroth index band, use the band/time column to create an explicit array of absolute times, from the relative time hours in the future that band originally represented, and replace “time” column with that

5. Place tweaked DataArray object into a xarray Dataset under feature name “PM25Forecast” (meaning forecast of $PM_{2.5}$) and return that Dataset which now has specifications required for the creation of a GeographicalDataset object

B.3.4 BlueSkyParser

The BlueskyParser object is an implementation of the GenericParser object which overrides the convert_raw_to_dataset method with one containing the process for specifically converting a Bluesky NetCDF4 file into a GeographicalDataset object. This process is as follows:

1. Open the NetCDF4 file into an xarray Dataset object, using the standard xarray open_dataset function, which loads the file’s data with it along the index coordinates of ROW, COL, and TSTEP, and alongside meta data. The meta data contains the amount of latitude/longitude each ROW/COL cell holds and the objective time values in UTC strings of each TSTEP under the attribute TFLAG

2. Using the amount of latitude and longitude each cell represents under the attributes YCELL and XCELL respectively, and the center point lat/lon for both arrays under YCENT and XCENT, create 1D coordinate arrays of the objective latitudes and longitudes to replace the index coordinates of ROW and COL. Center is used rather than the origin point, to reduce the uncertainty that comes from the precision limitations on YCELL and XCELL. Latitudes are flipped over at the end of this process due to the nature of grids in programming languages. Low row indices mean high y position in grids and vice versa, so since the 1D latitude coordinate array starts as lowest to highest,

it is flipped to be highest to lowest, so that high latitudes are paired with low row indices meaning high grid placement, and low latitudes are paired with high row indices meaning low grid placement

3. Using the objective string time values under the attribute TFLAG, convert the strings to datetime objects to create a 1D coordinate array of objective UTC time.

4. Rename the index coordinates of the Dataset from "ROW" to "lat", "COL" to "lon", and "TSTEP" to "time" to have dimension names matching the specifications to create a GeographicalDataset object

5. Place the gridded data of the Dataset, into a new xarray DataArray object with the new coordinates of the 3 1D coordinate arrays, so that the grid is not marked out by explicit time, latitude, and longitude, instead of by index coordinates and corresponding meta data

6. Place the new DataArray object into a new xarray Dataset under feature name "PM25Forecast" (meaning forecast of $PM_{2.5}$) and return that Dataset which now has specifications required for the creation of a GeographicalDataset object

B.3.5 MODISAODParser

The MODISAODParser object is an implementation of the GenericParser object which overrides the convert_raw_to_dataset method with one containing the process for specifically converting a MODIS AOD hdf file into a GeographicalDataset object. This process is as follows:

1. Load the MODIS AOD hdf file into an xarray Dataset object, via xarray's open_dataset function, specifically using the engine "pynio" which is required for NASA LAAD's specific hdf file format (the file could not be loaded by standard NetCDF3 or NetCDF4 libraries). Pynio does not have a Windows OS port so WSL will be required to use this parser, if on a Windows system

2. Assign the "Latitude" and "Longitude" attributes explicitly as coordinates for data, instead of just being another attribute in Dataset, and with names "lat" and "lon" per GeographicalDataset specifications

3. The Dataset attribute "Corrected_Optical_Depth_Land" contains three distinct measurements in its DataArray at 0.47, 0.55, and 0.65 microns, so separate those three out from the single DataArray into three separate DataArrays

4. Extract the DataArray's of attributes
   "Corrected_Optical_Depth_Land_wav2p1" and
   "Mass_Concentration_Land" as well

5. With these 5 DataArrays objects being separated out, store them in a new xarray Dataset object under the feature names
   "Corrected_Optical_Depth_Land_Solution_3_Land_47",
   "Corrected_Optical_Depth_Land_Solution_3_Land_55",
   "Corrected_Optical_Depth_Land_Solution_3_Land_65",
   "Corrected_Optical_Depth_Land_wav2p1", and
   "Mass_Concentration_Land".

6. All measurements occur at the same time so using PVL, which the meta data is written in, extract the time that the measurements were measured at, and use that single entry as the "time" coordinate for the Dataset

7. Return the Dataset which now has the specifications to create a GeographicalDataset object

B.3.6 MODISFRPParser

The MODISFRPParser object is an implementation of the GenericParser object which overrides the convert_raw_to_dataset method with one containing the process for specifically converting a split MODIS FRP files into a GeographicalDataset object. During the splitting process, the saved split hdf files are already put into a format fulfilling the GeographicalDataset specification, so the files are simply loaded into xarray Dataset objects and returned

B.4 Cleaning Step

B.4.1 GenericCleaner and the General Cleaning Process

The cleaning process is essentially the process of crunching data from a single dataset's multiple raw data files, into a single object containing standardized space grids, along standardized points in time, for a set number of features. This process is necessary as in the amalgamator, when producing pytorch tensors, it is necessary to draw space grids for each feature at a certain matching time across all datasets, and this would not be possible without crunching that data down into standard known spaces and times. This process essentially boils down to two core steps when looking at it at

the highest level possible. The first step is to find and select all of the raw data files that contain data which is to be put into the resulting grid. The second step is to convert those raw data files into that grid.

The GenericCleaner is an abstract object which all other cleaners extend, and one that contains the method create_featuretimespacegrid which contains the two step process that was just mentioned. The GenericParser contains only the implementation for the first step however. Any object fully implementing GenericCleaner will need to include an implementation of the abstract attributes file_name_regex, file_name_datetime_regex, and file_name_datetime_fmt. GenericCleaner uses these attributes, along with a given file directory and a given data time window, to get the paths to all the raw data files that are to be included, using the get_files method, which is its implementation of the first step. It is important to note that files must include a datetime string in their name, in order to be properly selected by the get_files method. This file selection on date is necessary as real time predictions using the model will require training on files released at previous times to the prediction, thus requiring us to use the cleaners to create space time grids of present times, using data files from earlier times (so data window selection is necessary since the grid times can be offset from the data file's specific marked time). More specifically in the case of forecasts, the files do not have a single measurement for a single time, but rather multiple forecasts can forecast a single time, so data range selection on file name date is necessary to choose which forecast day to use, or to select all of them. The get_files method functions as follows:

1. Grab all files names matching the file_name_regex attributes from the given folder
2. Grab all the marking dates of the files from their file names using file_name_datetime_regex to search for it and file_name_datetime_fmt to parse it into a datetime object
3. Filter, only keeping files within the given data time window
4. Create absolute paths out of those files names, and return that list of paths as strings

The list of absolute file paths is then passed into the abstract method convert_files_tofeaturetimespacegrid which will contain the processing second high level step, and will be implemented by the extensions of the GenericCleaner. This method will require the crunching down of data across time

and grid space, as well as information on which features will be included, how to load the file, and if the space 1D axes require meshing which is why the abstract attributes of time_cruncher, cell_cruncher, expected_features_array, parser, and requires_mesh must also be implemented by anything extending the GenericCleaner. The GenericCleaner is located in smoke/smoke/clean/cleaners.py.

B.4.2 Cleaning Tools

The cleaners have two tools available to them, which are located in smoke/smoke/clean/toolset.py. These two cleaners are the TimeCruncher and the CellCruncher, which respectively crunch data along the time axis and crunch data along the space axis, in order to crunch them down to standardized unique points along the axes of the spacetime grids. Both were created as abstract objects in order to make the functionality of how they crunch down one group of data modular, such that depending on the dataset one could easily swap it out for another.

The CellCruncher is an abstract object which is meant to crunch data down spatially, such that in the space grid every single grid cell has at most one data assignment. This is necessary as overlapping assignments, if not dealt with, will cause data to just be replaced which will incorrectly represent data in the final grid product. Its main method crunch_data performs this action, taking in an array of paired grid assignments and an array of corresponding data, and crunching them outputting a masked array of unique grid assignments and a corresponding array of crunched down data. It also has the secondary functionality of filtering out any invalid assignments and the data associated with them. It does all of this through the following process:

1. Data and grid assignments are flattened in case of any 2D shaped assignments/data being given to it. 2D shape will not be preserved at the end due to similar assignments being crunched to one number, and non assigns being thrown out, so doing this at the front makes the process much easier

2. numpy.ma.array has a bug where the boolean array of the mask can come out as just a single entry, so if this happens this step just rebuilds the array to be the same size as the assignments as it should be. This only happens when the assignment is a numpy.bool_ of False.

3. Filter out any invalid assignments and corresponding data as the mask dictates, which leaves us with a standard numpy array for both assignments and data

4. Check that there are any assignments left after filtering, and skip the rest of the process if all assignments had been filtered out

5. Find how many times each pair assignment occur

6. Using complex numbers turn the pair assignments into a single number, and filter out any (including data) that only occur once as they do not need to be crunched. This will save time

7. For the rest which have overlapping data in single cells crunch all in the same cell down using the abstract method crunch_similar_cells which will return a single data entry for that cell

8. Return the list of now unique grid assignments and the corresponding data

The crunch_similar_cells method is the swappable functionality of the CellCruncher, as depending on the dataset one might want to do a specific operation on overlapping data such as taking a sum for something like overlapping smoke clouds, or taking a mean for something like $PM_{2.5}$ measurement. The implementations of CellCruncher contain the functionality for this, with the two existing ones being MeanCellCruncher, which returns a mean of everything overlapping, and SumCellCruncher, which returns a sum of everything overlapping.

The TimeCruncher is an abstract object meant to crunch down similar spatial grid data on time from an arbitrary number of points on a time coordinate axis, to a regular standardized evenly spaced time coordinate axis. It takes in two TemporaryTimeSpaceGrid (TTSG) objects as arguments, one of which is populated with data at the arbitrary number of points along a time axis, which will be called the original TTSG, and one which is unpopulated with data, but has a standardized evenly spaced time axis within it which will be called the target TTSG. It also takes an argument of time bin size of the target TTSG in hours for sake of explicit setting so as to not have to extrapolate from the time coordinate axis of the target TTSG. The TimeCruncher essentially functions to take the data from the original TTSG and crunch it down to fit into the standardized time of the target TTSG, which it does through it's crunch_to_result_TTSG method. This method does this through the following process:

1. Grab the time axis of the original TTSG and the time axis of the target TTSG
2. Check to make sure that there are populated grids in the TTSG, if not just return the empty target TTSG as there is no valid values to populate it with
3. For every time on the time axis of our target TTSG, gather all of the space grids with times between time and time - time bin size from the original TTSG and crunch all of them to a single space grid for that time using the abstract method crunch_to_single_grid
4. Populate those points in the target TTSG with the crunched grids and return the target TTSG

The crunch_to_single_grid is the swappable method for the TimeCruncher and is implemented by any full implementation of TimeCruncher. The implementation will contain the process necessary to turn an arbitrary number of space grids into a single space grid for a given time. The only implementation so far in the code is the AvgTimeBinTimeCruncher which simply takes a mean across time of the space grids given, however there are multiple other functions that can be added for different ways of crunching those space grids.

To expand the toolset and create additional functionality for either the CellCruncher or the TimeCruncher one must simply create an implementation of either, replacing the crunch_data or crunch_to_single_grid method respectively. The new cruncher can then be added to the desired cleaner, under the abstract attributes time_cruncher and cell_cruncher, to incorporate the new functionality into the cleaning process.

B.4.3 GeneralConversionCleaner

The GeneralConversionCleaner is located in smoke/smoke/clean/cleaners.py and is the primary implementation of the GenericCleaner class' convert_files_tofeaturetimespacegrid method (i.e. contains the primary and most general process for step two of the high level process, which is converting a group of selected raw data files into a populated standardized 4 dimensional grid). For this process the GeneralConversionCleaner requires arguments of a smoke.box.Box.Box object which will handle grid assignments, and is explained further in the B.7.1 Box subsection, an exclusive start time for our standardized grid, an inclusive end time for our standardized grid, a time resolution for our standardized grid, and the list of GeographicalDataset loadable files which were found using the get_files step.

The GeneralConversionCleaner also makes use of the abstract attributes of time_cruncher, cell_cruncher, expected_feature_array, parser, and requires_mesh as stated in the B.4.1 subsection regarding the GenericCleaner. The GeneralConversionCleaner then carries out the conversion of all those files, into a 4D grid within the object of a smoke.box.FeatureTimeSpaceGrid.FeatureTimeSpaceGrid, which has the coordinate specifications spatially of the given Box, and temporally of the given start, stop, and resolution parameters, via the following process:

1. A GeographicalDataset object of data from each of the given raw data files is loaded using the abstract attribute of parser, so that all data from the files can be accessed with the standardized wrapper functions

2. The endpoint FeatureTimeSpaceGrid (FTSG) is created using the given Box to specify the space grid, and the given start, stop and time resolution parameters, to specify the time axis, with one of these 3D spacetime grids being created for every feature in the expected_feature_array given under the abstract attribute

3. For every feature in the endpoint FTSG the 3D spacetime grid is populated via the following process:

    (a) Every xarray DataArray within each GeographicalDataset of the feature is extracted. Then for every single time on every time axis the lat axis, lon axis, and 2D data along space are extracted from each of these DataArrays

    (b) For every time the 2D data is then flattened and given an assignment in the space grid of the given Box using the Box's functions, based on the latitude and longitude axes for that data. This is where the requires_mesh attribute is utilized as the latitude and longitude axes are meshed to match the 2D data, if they start as 1D arrays

    (c) Then for every unique time in all times gathered in the step (a), all of the data and corresponding grid assignments that have that time are grouped

    (d) For every one of those unique times and grouped data/assignments the cell_cruncher attribute is called to crunch down all data and assignments for a unique time, into a unique set of data and grid assignments such that there is a single uniquely populated space grid for each

(e) Then using every unique time and crunched down space grid, a TemporaryTimeSpaceGrid (TTSG) is created and is populated with all the data, and another unpopulated TTSG is created of the resulting FTSG's current feature's 3D spacetime grid. Using the attributes of the time_cruncher the TTSG, containing all the data, is then crunched down to the target TTSG.

(f) Now with a TTSG containing the 3D spacetime grid for the feature and containing relevant data of that feature from every data file, the feature's empty 3D spacetime grid is replaced with the one from this TTSG, thus populating the endpoint FTSG's feature's 3D spacetime grid

With that all completed for every feature of the FTSG, a fully populated 4 dimensional grid across features, time, and space will have been populated, and thus can be returned completing the implementation of the highest level second step. This GeneralConversionCleaner's process will work for any arbitrary amount of GeographicalDatasets from a single dataset, but will take an extremely long time if the data given is not sparse. It will still work, but depending on the assumptions made, this process can be sped up.

B.4.4 ConsistentGridCleaner

The ConsistentGridCleaner is located in smoke/smoke/clean/cleaners.py and is an extension of the GenericConversionCleaner class. The purpose of the ConsistentGridCleaner class is to take advantage of the assumption that the dataset that is being cleaned has a consistent space grid across all of its raw data files, by using this assumption to speed up the normal GenericConversionCleaner class' convert_files_tofeaturetimespacegrid process. It does this by overriding 2 specific steps in the process which are as follows:

1. For step 3.b in the B.4.3 GenericConversionCleaner's process, the step is overridden such that in the GenericConversionCleaner's 2D data grid assignments, only for the first time in all times is the grid assigned instead of doing so for every single time. This can be done since all other data grids will end up getting the same assignments, under the consistent space grid assumption, and thus saves time as only one of these grids has to undergo the process. The one grid assignment is then assigned to every other time, matching the end state of 3.b in the B.4.3 GenericConversionCleaner's process.

2. For step 3.d in the B.4.3 GenericConversionCleaner's process, the step is overridden such that all data 2D grids in a single time which are to

be crunched, are simply crunched across all of their time axes with a numpy.nanmean before entering any CellCruncher. This can be done since the assumption of a consistent space grid is made across all 2D data grids, so crunching across time first will simply mean doing MeanCellCruncher's job in one single array step. This saves time and is absolutely necessary as normally the CellCruncher has to iterate over every single repeated cell assignment, and in a consistent grid system if there is more than one grid for a time, every single one of those cells will have a repeated assignment meaning a huge number of steps are necessary. The single crunched grid is then pumped into the given CellCruncher, with the first grid of assignments (since all are the same under the assumption) in a similar fashion to the end of the step for 3.d leaving it in a similar state at the end. It is important to note that ConsistentGridCleaner's use of the numpy.nanmean in this step means that the crunched data will be only valid if the cell_cruncher desired for this cleaning was a MeanCellCruncher. One can make another ConsistentGridCleaner under a different name, with a different operation across the time axes replacing numpy.nanmean to properly get the functionality for another CellCruncher.

Other then these two overrides however, the ConsistentGridCleaner uses the processes in the GeneralConversionCleaner as described in subsection B.4.3.

B.4.5 Dataset Specific Implementation of Cleaners

The final implementation of the GenericCleaner class requires the implementation of both the set of abstract attributes and the convert_files_tofeaturetimespacegrid method. With the GeneralConversionCleaner, ConsistentGridCleaner, or some additional equivalent class, the convert_files_tofeaturetimespacegrid method can be implemented, requiring just one more additional extension of those classes to fully implement the GenericCleaner. The things left to implement, the abstract attributes, will be specific to a dataset as file names, parsing, and the types of crunching desired, will entirely be contingent on how that particular dataset's files are.

When extending the GeneralConversionCleaner, ConsistentGridCleaner, or some additional equivalent class to get to a full implementation of GenericCleaner, the following attributes must be implemented which are specific to the particular dataset meant to be cleaned by the cleaner:

- parser: the parser to use when loading the dataset's files into GeographicalDatasets for the cleaner
- expected_features_array: the array of names for feature DataArrays contained in each of the resulting GeographicalDatasets from the parser
- file_name_regex: Regular expression which will match the file names of the files from this dataset
- file_name_datetime_regex: Regular expression to find the datetime string contained in each of the files' names for this dataset
- file_name_datetime_fmt: Python datetime strptime format to parse the datetime string contained in the files' names
- time_cruncher: TimeCruncher object to use to crunch data overlapping temporally
- cell_cruncher: CellCruncher object used to crunch data overlapping spatially
- requires_mesh: True if dataset's GeographicalDataset has a 1D axis for latitude and longitude that needs to be meshed to match the size and shape of 2D data given per time

Currently in the data pipeline the full implementations that are written are the FireworkCleaner, BlueSkyCleaner, MODISAODCleaner, and the MODISFRPCleaner which contain the abstract attributes necessary to clean the files from the datasets of their namesake. The FireworkCleaner and BlueSkyCleaner both implement the ConsistentGridCleaner since they contain non-sparse data that occurs always on a consistent spatial grid of longitudes and latitudes, and the MODISAODCleaner and MODISFRPCleaner implement the GeneralConversionCleaner since they contain sparse data and don't have a consistent spatial grid. Each of these cleaners uses a cell_cruncher of a MeanCellCruncher and a time_cruncher of a AvgTimeBinTimeCruncher since any overlapping data does not necessitate more of that value in that area (requiring a sum), but rather a different measurement in that area (requiring a mean). FireworkCleaner, BlueSkyCleaner, and MODISFRPCleaner all have requires_mesh set to True since they all have 1D coordinate axes for latitude and longitude following their parsing, and MODISAODCleaner has requires_mesh as False since it has a 2D grid for latitude and a 2D grid for longitude post-parsing. Each uses the parser from smoke.load.parsers of their namesake and the expected_features_array

that result from that parsing process which all are mentioned within B.3. Finally, the implementation of file_name_regex, file_name_datetime_regex, and fine_name_datetime_fmt are set to the specifications of the names given to files for that dataset when downloaded by our download suite, so any changes there will require changes to these attributes in the cleaners.

B.4.6 Running Cleaning Process

To extract data from raw data files into output FeatureTimeSpaceGrids by running the cleaners one must use the smoke/smoke/clean/run_cleaners.py script to run it, after editing the smoke/smoke/clean/run_cleaners_config.yml or a copy of it at a known path. To run the script using the edited config yaml file, one must simply call python on the script with the path to the config as an argument. An example command line call for running this script with a working directory of smoke/smoke/clean is shown below:

```
python run_cleaners.py <path to run_cleaners config yaml>
```

The yaml file contains settings for the output FTSG's space grid size in km, the time range as defined by a start and stop time in ISO-8601 (each day in the time range will have a FTSG generated for it for all datasets set to run), and a block of settings specific to each dataset. The dataset block contains 4 attributes common to all blocks of settings for datasets which are as follows:

1. run: Which tells you whether to run this dataset through the cleaners or not
2. file_directory: The directory to look in to find this dataset's raw data files
3. output_directory: The directory to save out FTSGs to for this dataset
4. grid_time_res_h: The time resolution to use for the daily FTSGs generated

For firework and bluesky, the two forecast datasets, there are also multiple dataset blocks underneath them with three additional attributes each dataset block. Each of these dataset blocks defines a set of settings for one FTSG creation run under the forecast, with multiple per forecast being possible due to the multiple dataset blocks. There is also an additional attribute for each forecast called time_we_at_stand_before_grid_h which defines in hours the buffer zone between the start of any FTSG's day and the

time range valid to look for forecasts files within. So for example if there is a standing time of 1, when creating an FTSG for any of the dataset blocks below, anything past 1 hour before the FTSG's start, are files that have data that can not be used for that FTSG. In the multiple dataset block's three additional attributes, two of the attributes are set to define a data window to look for files within for that dataset's FTSG run, and are called data_window_end_n_hours_before_standing and data_window_size_h. These respectively define the end of the data window for that FTSG run relative to our aforementioned standing time, and the size of the data window in hours. Finally the third attribute of file_prefix just allows differentiation in name for FTSGs of that particular setting FTSG run vs. the other FTSGs from FTSG runs of other dataset blocks under that same forecast. This was essentially made as firework and bluesky can have multiple forecast files which contain forecast data for a single time, so by doing this you can generate one set of FTSGs for the closest forecast to the FTSG, and another for the 2nd closest to the FTSG, and so on. At this moment, firework has 4 since it can go up to 72 hours into the future in recent geotiffs meaning up to 4 forecasts for a time, and bluesky has 2 since it can go up to 50 hours in the future meaning up to 2 forecasts for a time.

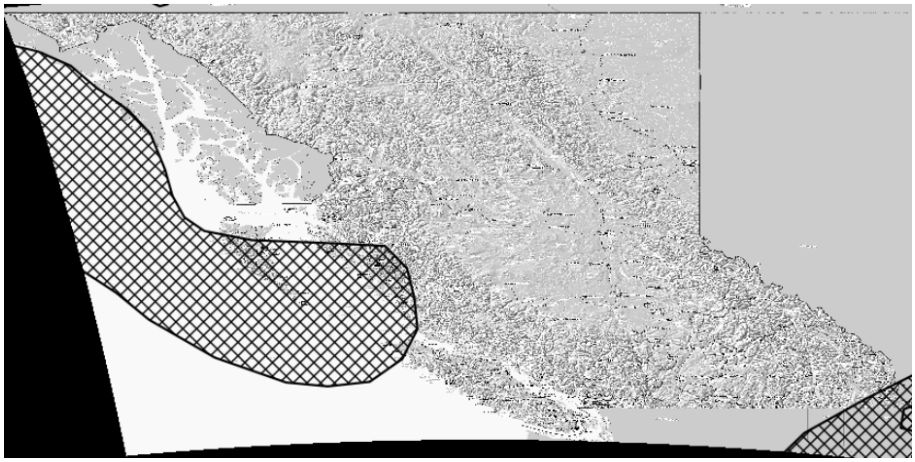
The actual run_cleaners.py script, when called, creates an FTSG for every day in the time range, loading data from the file directory given, and saving our FTSGs to the output directory given, depending on all of these attributes set in the given yaml file for each dataset. It is set currently to clean firework (and the 3 other closest forecasts), bluesky (and its other closest forecast), modis FRP, and modis AOD, but additional calls to other additional cleaners can be added in a similar fashion. It will only attempt to clean and produce FTSGs for each of these datasets, if they are set to run in the config file. Basically, the process for this script is that it generates a smoke.clean.cleaners.BCBox of the given km resolution and a date range for every day in the time range given. It will then for each of the datasets set to run, create an FTSG of every day in the time range for that dataset. If no files are given on a particular day, the FTSG will still be made, but it will be entirely filled with nan values.

B.5 NOAA Processing

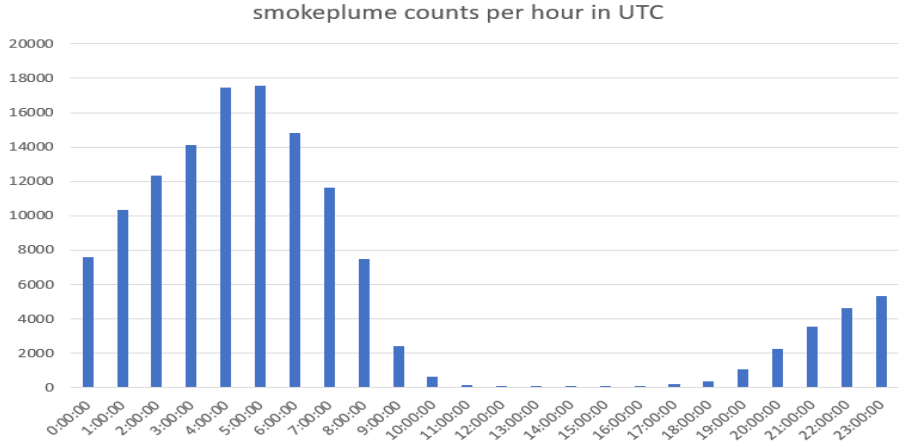
B.5.1 Exploratory data analysis

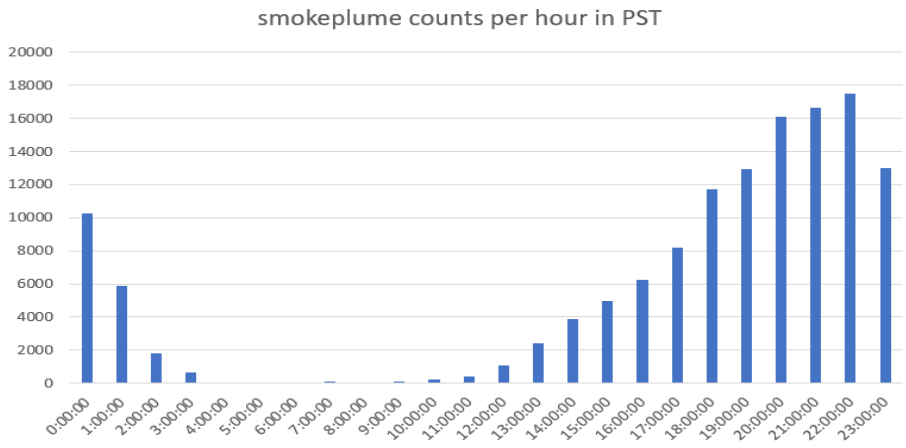
Below is a visualization of data from NOAA dataset. The underlying image is a raster of British Columbia, layered with a polygon shapefile of smoke

plume drawn.



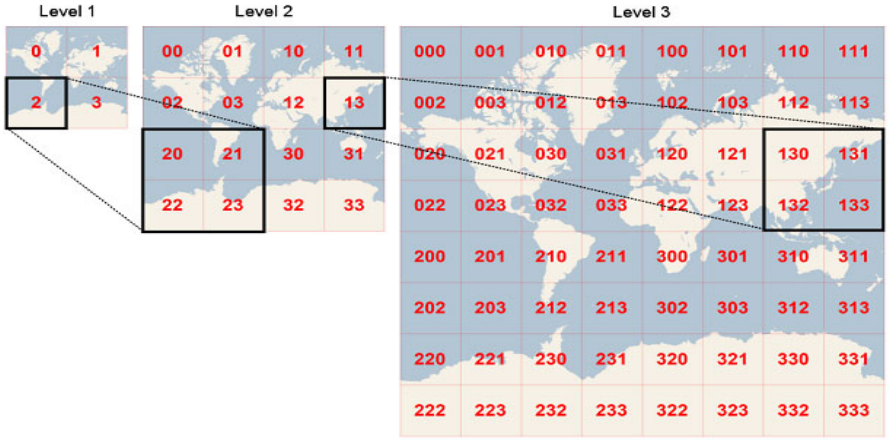
From a time series analysis of the number of smoke plumes drawn in a year, we found that smoke plumes are rarely drawn between the hours of 10:00 and 19:00 UTC (or 3:00 to 12:00 PST). We speculated that perhaps the smoke plume analysts do not work 24/7. However, after corresponding with the NOAA staff, we realized that smoke plumes are not drawn during these times primarily because of low visibility. Hence, instead of labelling each cell as 0 (no smoke) and 1 (smoke), we decided to introduce a third label (NaN) to indicate to the model that no data was collected but that does not necessarily mean there was no smoke.





Additionally, the smoke plumes are drawn at irregular time intervals, and sometimes erroneous start and end times are recorded (where end times come before start time). We decided to floor the time per hour and combine smoke plumes at that hour.

In order to extract binary features that indicate whether a cell in a grid of BC has smoke or not, we are using a geohashing approach illustrated below:



Imagine the world is divided into a grid with 32 cells. The first character in a geohash identifies the initial location as one of the 32 cells. This cell will also contain 32 cells, and each one of these will contain 32 cells (and so on repeatedly). Adding characters to the geohash sub-divides a cell, effectively zooming in to a more detailed area.

The precision factor determines the size of the cell. For instance, a precision factor of one creates a cell 5,000km high and 5,000km wide, a precision factor of six creates a cell 0.61km high and 1.22km wide, and a precision factor of nine creates a cell 4.77m high and 4.77m wide (cells are not always square).

Geohashing is used to turn each cell in the BC grid into a unique geohash

that can be compared with the smoke plume polygon to indicate whether that cell has smoke or not.

B.5.2 Geohashparser

Class Geohashparser will take parameters 1) source polygon shapefile directory, 2) individual shapefile, 3) results filepath, and 4) hours of day to exclude, and output a matrix of BC with cells assigned 1 for smoke, 0 for no smoke, and -1 during the hours in which smoke plumes are rarely drawn.

Please note that the bounding box of BC as well as geohash precision were set as static variables.

We check the polygon shapefiles to identify if the shapes are within the bounding box coordinates of BC, and compute geohash tiles from polygons that are within the box.

B.5.3 BoxMapper

In order to integrate features from different datasets, Class Amalgamator was created. An arbitrary 2D grid of BC is used ubiquitously by all datasets to map features into the same BC grid. Hence, Class BoxMapper was created to integrate the smoke plume feature into this ubiquitous BC box for each hour of each day from smoke plume data in 2018-2019.

An instance of the ubiquitous BC box (250x250) is created for each hour of each day, and if the hour falls under the “hours to exclude”, then the box is filled with -1’s. If the hour falls outside of the “hours to exclude”, then the box is filled with 0s to begin with. We then take the geohash output (lat/lon coordinates) from the Geohashparser and map it to the common BC box. A geohash cell that resides within a smoke plume polygon are assigned a value of 1, then the geohash cell coordinates are compared with the coordinates in the box to be mapped to the corresponding cell in the box.

B.6 BC MoE $PM_{2.5}$ Label Processing

B.7 Grid Representation

B.7.1 Box

B.7.2 BCBox

The BCBox is located in smoke/smoke/clean/cleaners.py and is a type of Box (from smoke/smoke/box/Box.py) which has a hard coded consistent set of input parameters for top left corner lat and lon, bottom left corner lat

and lon, and grid side distance in km. It still requires an argument of grid resolution however, which allows the changing of grid resolution during the cleaner process. In order to change the location of the space grid used to create the FeatureTimeSpaceGrids during the run_cleaner.py process, one must change the coordinates and distance settings passed into Box within this BCBox. This was created so that all grids created during the cleaning process would have the same space grid, and thus during the amalgamating step the end result tensors would be different feature measurements stacked on top of one another in a similar space grid.

B.7.3 FeatureTimeSpaceGrid

The FeatureTimeSpaceGrid (FTSG) class is a data storage object which extends the 2D theoretical grid representation of the Box from the smoke/smoke/box/Box.py file, into a physical populatable 4D grid representation within this object. This object does this by containing a 4D numpy grid of the axes in order of feature, time, row and column. The feature axis size is determined by the feature array given on creation to the FTSG and the time axis size is determined by the given exclusive start, inclusive end, and resolution parameters. These allow the FTSG on creation to create a time axis given the specifications, and creates a 4D grid with its 2nd dimension of time being the same size as that time axis. Finally the last 2 axes of row and column are determined by the given smoke.box.Box.Box object given to the FTSG, as the Box object is a theoretical grid, so the FTSG simply creates a 2D space grid which is of its specifications. The FTSG also has functions to save itself out in its current state into a tar.gz file containing all necessary data to rebuild that FTSG, and contains a function called load_featuretimespacegrid to load one of these saved tar.gz FTSGs back into a FTSG object in python. This is necessary as the cleaners must save out FTSGs and the amalgamator must load those FTSGs to pull slices from, in order to create the final pytorch tensors. The FTSG class is located in smoke/smoke/box/FeatureTimeSpaceGrid.py

B.7.4 TemporaryTimeSpaceGrid

The TemporaryTimeSpaceGrid (TTSG) is essentially a temporary object which extends the 2D theoretical grid representation of a smoke.box.Box.Box into a 3D physical grid with the addition of a time axis. This class is located in smoke/smoke/box/FeatureTimeSpaceGrid.py and is basically purely for use in the TimeCruncher tool in order to efficiently pass a bunch of 2D space grids along a third axis of an arbitrary sized, arbitrary ordered time

axis into the TimeCruncher and to store post time crunched data efficiently. Both of these are made a ton easier by having a poputable 3D grid of Box specifications and a flexible time axis, which is why the TTSG class was created.

B.8 Amalgamator

B.8.1 Overview and Process

The Amalgamator is the object that carries out the pytorch tensor creation, creating tensors from 2D grid space slices drawn from each feature in saved out Firework, Bluesky, MODIS AOD, and MODIS FRP FeatureTimeSpaceGrids (FTSG) and from saved out NOAA .npy files post NOAA branch processing. The Amalgamator is located in
smoke/smoke/amalgamate/Amalgamator.py. Upon creation the Amalgamator requires arguments for the multiple directories containing the $PM_{2.5}$ labels, the Firework, Bluesky, MODIS AOD, and MODIS FRP saved out FTSGs following the processing of run_cleaners.py, and the NOAA grids following processing of the NOAA branch. If any of these datasets will not be included in the finalized tensor, the arguments can be just left as None. The Amalgamator contains methods named firework, bluesky, modisfrp, modisaod, and noaa which are each meant to load the 2D space grids of all features within the datasets of their namesake, from the directories given on Amalgamator creation, for a given time called label_time. This label_time is meant to be the time of the ground truth label that the result tensor is going to try and predict, so each of these methods contains the time logic for finding which grid slice to use to try and predict that specific time. Each of these FTSG methods has a process based on the output FTSG of the current run_cleaners.py process explained in B.4.6, so they are all reliant on FTSGs being saved out as whole days as they are in the process, but can handle any of the flexible parameters given in the run_cleaners_config such as a changed resolution in time or space.

The methods of forecasts firework and bluesky simply grab the direct corresponding time of label_time from their respective FTSGs, as there is no necessary time logic to simulate real time due to the FTSGs being of a forecast which is a measurement representative of the future. For example if we were standing at 0600 hours and wanted to predict what would happen at 1200 hours with a forecast, we would just use whatever the forecast is forecasting at 1200 hours. This is also largely due to the time logic offset for simulating real time for the forecasts, being taken care of during the

run_cleaners.py script, as the buffer time and data window for real life daily predicting is set by those settings in the cleaning process. The physical measurements of modisfrp, modisaod, and noaa are all represented in the FTSGs and saved out .npy files at the exact time they were measured, so in order to simulate a real time prediction some additional time logic is required. For example if we were standing at 0600 hours and wanted to predict what would happen at 1200 hours with a forecast, we would need to use physical measurements sometime before 0600 hours as we don't have physical measurements in the future. At the moment all three methods have just logic similar to the forecasts of grabbing the physical measurements at the given label_time for simplicity for the proof of concept model. However if one were to simulate real time, and try and train the model to do real time predictions one would have to alter the method calls. Modisfrp and modisaod both contain a optional parameter of simulate_real_time which can be flipped to True to have them simulate real_time physical measurement difficulty. They do this by, once flipped, using the setting of daily release time for either dataset, and instead of grabbing directly the corresponding label_time, grabbing the closest previous release time of physical measurements to that label_time. Noaa does not have a simulate real time option, so that would have to be written for it once the model gets to that stage.

The actual tensor creation using all of these methods takes place in the method make_pytorch_tensor which, given a label_time, will generate a prediction tensor to try and predict the label at that given label_time. Alongside the time to create a prediction tensor, it takes in an argument of a dataset list which is a list of string containing any combination of the strings "firework_closest", "firework_2ndclosest", "firework_3rdclosest", "firework_4thclosest", "bluesky_closest", "bluesky_2ndclosest", "modisaod", "modisfrp", and "noaa". These strings all correspond to a single possible FTSG creation dataset set from run_cleaners.py. When the method is run for a given label_time first the Amalgamtor checks to see if there is a ground truth label of label_time existent in the $PM_{2.5}$ labels folder using the logic in the method check_pm25_PST_file_exists. It will then iterate through the dataset list drawing a 2D space grid for each dataset contained within it, using the methods previously mentioned, and checking to see that valid prediction data is given. If valid prediction data from any of the datasets is not available for that time, an error is raised to prevent an invalid prediction tensor from being made. If all 2D space grids of data returned are good, all 2D space grids are stacked together into a tensor then saved out under a unique name to the particular label_time. This resulting tensor will be 3

dimensional, with a first dimension of feature measurement, and following dimensions of 2D space. The resulting pytorch tensor will be a prediction tensor containing data relevant to predict a label of similar time, using the CNN. It is important to note that the Amalgamator does not check if the grid km resolution is the correct size or not, so all directories containing FTSGs and .npy files of saved out prediction data must be in the same resolution of the same space grid (easiest if all are generated using the same BCBox object as that will have consistent space grid coordinates and grid resolution).

B.8.2 Running the Amalgamator

To run the amalgamator using already made code the script run_amalgamator.py in smoke/smoke/amalgamate is available to use. run_amalgamator.py when run will generate pytorch tensors containing certain datasets for a time range, both as specified in a config yaml file, saving the tensors out into a given output directory path. To run the file first one must edit the config yaml to desired specifications which can be the smoke/smoke/amalgamate/run_amalgamator_config.yml file or an equivalent copy in some other known place which contains the same attributes. To run the script using the edited config yaml file, one must simply call python on the script with the path to the config as the first argument and the path to the desired tensor output directory as the second argument. An example command line call for running this script with a working directory of smoke/smoke/amalgamate is shown below:

```
python run_amalgamator.py <path to run_amalgamator config file>
<path to output directory for created tensors>
```

As the Amalgamator itself is based only on datasets run_cleaners.py is hard-coded to clean, the run_amalgamator.py script and accompanying config yaml also contain settings and processes for adding only those datasets which are the 4 closest firework forecasts, 2 closest bluesky forecasts, modis FRP, modis AOD, and noaa. The config yaml contains relatively self explanatory settings. The datetime start, stop, and resolution parameters set the times in UTC which the run_amalgamator script will attempt to create tensors for. The dataset directories are simply the directories that one has saved their FTSGs or npy grids into during the cleaning process, and where the time labelled $PM_{2.5}$ ground truth labels are. And the rest of the attributes are simply the names of the datasets that can be included in the tensor, and should be set to True or False depending on if the dataset is to be included in the generated tensors or not.

Upon running the script, the settings for datetime start, stop, and time resolution, will be formed into an array of those settings. An Amalgamator object will be created using the dataset directory settings loaded from the config. Then for every single time in that time array, the amalgamator will attempt to generate a pytorch tensor of prediction data, which is meant to predict the label at each time given. As stated before, the Amalgamator will return an error when the labels for that UTC time do not exist, or if any of the datasets being added have an invalid data grid of prediction data for that time. In any case of this, the tensor creation for that particular time will be skipped, with the log stating which dataset/label caused the skip. The result then after running the script should be as many prediction tensors as were valid in the time range specified, saved out into the given output directory.

B.9 Validation

The pipeline has been validated by the testing suite located in the smoke/tests directory with amalgamate, box, clean, load, and split testing the components kept in the folders of their namesake, located in the smoke/smoke directory. The tests in amalgamate, box, load, split and the test_CellCruncher.py and test_TimeCruncher.py scripts can all be run to confirm the function of their respective components, which will be useful for testing core functionality upon any changes to any of the classes.

The one test that is not as straightforward is the manual visual testing located in smoke/tests/clean/test_cleaners_manually.py which creates animations based on a set of saved out FTSGs and the initial data that should be going into it post running of a cleaner. Each animation will contain various plots of all data files that should be going into a single FTSG and the actual resulting grid of the FTSG across all times in the FTSG. To use this animation to diagnose the cleaners, one must simply check that the information being displayed that exists in the saved out FTSG makes sense visually with the initial data that is being displayed beside it.

To run the script to generate these animations one must simply edit the config.yml file in smoke/tests/clean or a copy of the config file in some known location, to specify the output directory for animations, the location directory of the post cleaning FTSGs to test, the directory containing raw data files that went into that FTSG, the data window specified to search for raw data files for a particular FTSG during the cleaning process, and which cleaner is being tested. For now the animations can only be generated for Firework,

Bluesky, MODIS AOD, and MODIS FRP as those are hardcoded but one can edit the script and config to add additional tests for additional cleaners. Finally one would simply call from the command line the script with an argument of the config.yml path.

B.10 Recommendations

If one were to expand the model training to additional datasets and wanted to use pipeline infrastructure to add this type of data into the tensors, there are three places which one would definitely have to edit in the core of the pipeline. First, one would have to add an additional parser, which contains the process for loading the dataset's file format into an xarray Dataset with specifications for the creation of a GeographicalDataset. Second, one would have to add an additional full implementation cleaner, which can extend either the GeneralConversionCleaner or ConsistentGridCleaner or additional cleaner depending on the data in the dataset. This full implementation cleaner would have to implement the various abstract attributes which will be specific to the additional dataset. Finally one would have to add an additional function to the Amalgamator to load space grids for that dataset, and a call to that function in make_pytorch_tensor. This would fully implement the dataset into the core functionality of the pipeline.

As per the running of the pipeline accounting for this new dataset in either run_cleaners.py or run_amalgamator.py it will be a decent bit harder to add the functionality. One can simply follow the template of other datasets run through those scripts to add the consideration of the additional dataset to the code. However it is much more recommended, assuming one has a large chunk of time, to just remake the run_cleaners.py script, Amalgamator, and run_amalgamator.py script to deal with an arbitrary number of datasets and to not be tied to hard-coded non generalized processes. One while doing this, can use the current code functionality of each as a template for what processes need to be carried out for each dataset, and can then based on that example create a generalized way of doing so. Unfortunately, due to time limitations, this was not something we could implement in the pipeline, so it is as of right now tied to the specific datasets and functionality specifically that we required.