A Hitchhiker’s Guide to Anomaly Detection with Astronomaly

Michelle Lochner\textsuperscript{1,2} and Bruce. A. Bassett\textsuperscript{3,4,5}

\textsuperscript{1}Department of Physics and Astronomy, University of the Western Cape, Bellville, Cape Town, 7535, South Africa; mlochner@uwc.ac.za
\textsuperscript{2}South African Radio Astronomy Observatory (SARAO), 2 Fir Street, Observatory, Cape Town, 7925, South Africa
\textsuperscript{3}African Institute for Mathematical Sciences, 6 Melrose Road, Muizenberg, 7945, South Africa
\textsuperscript{4}Department of Maths and Applied Maths, University of Cape Town, Cape Town, South Africa
\textsuperscript{5}South African Astronomical Observatory, Observatory, Cape Town, 7925, South Africa

Abstract.

The next generation of telescopes such as the SKA and the Rubin Observatory will produce enormous data sets, requiring automated anomaly detection to enable scientific discovery. Here, we present an overview and friendly user guide to the Astronomaly framework for active anomaly detection in astronomical data. Astronomaly uses active learning to combine the raw processing power of machine learning with the intuition and experience of a human user, enabling personalised recommendations of interesting anomalies. It makes use of a Python backend to perform data processing, feature extraction and machine learning to detect anomalous objects; and a JavaScript frontend to allow interaction with the data, labelling of interesting anomalous and active learning. Astronomaly is designed to be modular, extendable and run on almost any type of astronomical data. In this paper, we detail the structure of the Astronomaly code and provide guidelines for basic usage.

1. Introduction

Modern and future telescopes including MeerKAT\textsuperscript{1}, the Square Kilometre Array\textsuperscript{2} and the Vera C. Rubin Observatory\textsuperscript{3} produce enormous quantities of data, making discovery of new and rare astrophysical phenomena impossible by manual data inspection. Machine learning has excellent potential for anomaly detection, but machine learning algorithms cannot distinguish between “interesting” anomalies and those that should

\textsuperscript{1}https://www.sarao.ac.za/science/meerkat/about-meerkat/
\textsuperscript{2}https://www.skatelescope.org/
\textsuperscript{3}https://www.lsst.org/
be ignored. Furthermore, unlike classification problems where class definitions are generally agreed upon between scientists, different people will find different anomalies interesting. For example: an artefact may not be of interest to an astronomer looking for strong lenses but may be interesting to a pipeline scientist in charge of data quality. To address this challenge, we developed Astronomaly (Lochner & Bassett 2021) which is a general purpose framework for active anomaly detection in astronomical data. Astronomaly is publicly available on github\(^4\). We leave details of the framework and analysis of its performance to Lochner & Bassett (2021) and focus here instead on the implementation.

2. Overview of Astronomaly

Astronomaly is designed to be modular and flexible. It can either be used as a pure Python library for anomaly detection, or it can be used with the frontend to aid in active anomaly detection. It operates on the principle of “code-as-config” which allows maximum flexibility to operate with many data types, with the downside of being more complex to use.

Figure 1 shows an outline of Astronomaly, indicating the different Python and JavaScript components. It can be seen that the Python backend follows a standard machine learning pipeline structure. First data is read into a standard format and pre-processed as necessary. The next step is feature extraction, where complex data is simplified and summarised down to a smaller set of numbers. This is a crucial step and the choice of feature extractor will largely dictate the types of anomalies the algorithm will be sensitive to. If necessary, further dimensionality reduction can then be applied followed by any necessary post-processing. The features are then finally ready for a machine learning algorithm to be applied to detect anomalies. Astronomaly also allows the user to visualise the high dimensional feature space in two dimensions using embedding tools.

After this pipeline has been run, the JavaScript frontend allows the user to view the data from most to least anomalous, interact with it, assign scores based on how interesting each object is, and applying an active learning algorithm to improve the anomaly detection.

3. Example Astronomaly script

Several example scripts are included in the Astronomaly package. Here we step through the example script galaxy_zoo_example.py, currently found in the scripts folder of the github repository. This section may be best understood by opening this script and following it along with the text.

A simple Python script is how a user interacts with Astronomaly. Anything can be done in this script, allowing a user to (for example) use external tools to extract features or manipulate data. The only stipulation is that the script must provide a run_pipeline() function which must return a dictionary containing (at minimum) the following keys:

\(^4\)https://github.com/MichelleLochner/astronomaly
Figure 1. Flowchart of Astronomaly software design. Rectangles indicate code components while the slanted parallelograms are outputs and inputs. The backend (blue) is built in Python while the frontend (green) is built in JavaScript, with a flask-based API connecting them (Lochner & Bassett 2021).

- 'dataset' - an object of type Dataset (further described in subsection 3.2)
- 'features' - a pandas (Wes McKinney 2010; pandas development team 2020) dataframe containing the final set of features, with the object identifier as the index and the features as columns (the column names will be assumed to be the feature names and used in the frontend)
- 'anomaly_scores' - a pandas dataframe containing the machine learning anomaly scores, with the object identifier as the index and the anomaly score as a column called 'score'
- 'visualisation' - a pandas dataframe containing the coordinates (i.e. two columns) for visualisation of the feature space, with the object identifier as the index
- 'active_learning' - an object that can run active learning on calling a function called 'run()' (see further details in subsection 3.6)

3.1. Settings
The first part of the galaxy_zoo_example.py script simply sets up the file paths for the data and the output. Astronomaly by default stores the output of each stage of
Figure 2. Screenshot of Astronomaly frontend interface displaying an example galaxy. The numbered buttons allow the user to grade how interesting each anomaly is, the drop-down box allows changing how the data are ordered and the retrain button will call the active learning algorithm from the backend to improve the anomaly detection. Information about each object is displayed on the right. The visualisation tab changes the interface to an interactive 2d visualisation of feature space.

the pipeline as parquet\(^5\) files. It will attempt to detect if a stage of the pipeline has already been run on the same data and, if it has, will load the stored output instead. This makes it easy to switch parts of the pipeline, such as experimenting with different machine learning algorithms, without needing to rerun slower parts such as feature extraction. This behaviour can be disabled at any stage of the pipeline by providing the keyword `force_rerun=True`. Deleting the automatically generated log file, `astronomaly.log` will also force a rerun of everything, since the log is what is used to detect any changes to pipeline stages.

### 3.2. Reading in the data

In the `run_pipeline()` function, usually the first step will be to load up the data. There are a variety of options in the `data_management` package, allowing reading in of images (fits files and normal image file types), time series data and raw data such as spectra. The dev branch of the github repository will generally contain the latest data readers available and it is straightforward to implement new ones (see section 5 and the online documentation for guidelines).

The line of code below reads in a set of image thumbnails into an object of type `Dataset`, which the rest of Astronomaly is able to work with. The key func-

\(^5\)https://parquet.apache.org/
tionality of objects of this type are access to an index variable, which simply contains the list of object identifiers in the dataset, and two functions get_sample() and get_display_data() which both take an index and return the data for that object. In the latter case, the function must return a version of the data ready to be displayed by the JavaScript frontend. The Dataset object must also contain a metadata attribute which is a pandas dataframe with additional data about each object.

```python
image_dataset = image_reader.ImageThumbnailsDataset(
    directory=image_dir, output_dir=output_dir,
    transform_function=image_transform_function,
    display_transform_function=display_transform_function
)
```

Notice two important keywords, transform_function and display_transform_function. These are functions or lists of functions that transform the data as it is read in. This is often a critical step in anomaly detection, allowing transforms like sigma clipping and non-linear flux transforms to be applied, highlighting critical features of the data to the algorithm. These functions will likely have a large impact on the success of your pipeline. In this script, they are defined earlier on and make use of a library of functions provided in the preprocessing package. To avoid large up front computation, these transforms are called only when get_sample() is called. A separate set of transform functions can be used for the visual display, to allow a user to, for example, see the original image instead of a sigma clipped image.

### 3.3. Feature extraction

After the data has been read in and pre-processed, the next step is to extract features. In this script, we use the same simple feature extractor used in Lochner & Bassett (2021), which fits a series of ellipses to isophotes and uses the parameters as features. The method, which relies on OpenCV (Itseez 2015), is very sensitive to how the data are pre-processed.

```python
pipeline_ellipse = shape_features.EllipseFitFeatures(
    percentiles=[90, 80, 70, 60, 50, 0],
    output_dir=output_dir, channel=0, force_rerun=False,
    central_contour=False)
features = pipeline_ellipse.run_on_dataset(image_dataset)
```

An object of type PipelineStage is created first. While this is not essential (a user could for instance read in a set of features extracted elsewhere into a pandas dataframe), inheriting from the Astronomaly base class allows automated logging, re-run detection and in the future, parallelisation. The parameters for any feature extractor are passed as keyword arguments to the initialisation of the object. run_on_dataset(), the function that actually performs the feature extraction, should only take a single argument, an object of type Dataset. There are a variety of feature extractors to choose from in the feature_extraction package.
3.4. Feature post-processing

In the galaxy_zoo_example.py script, we then choose to scale the features to be centred around zero with unit variance. This is standard practice in machine learning and is a critical step for many (but not all) algorithms. The process for post-processing features is almost identical as that for extracting them, except you pass the function a pandas dataframe as input.

```python
pipeline_scaler = scaling.FeatureScaler(force_rerun=False,
                                        output_dir=output_dir)
features = pipeline_scaler.run(features)
```

3.5. Machine learning

We then run a machine learning algorithm for anomaly detection. Again, an external algorithm can be used as long as a correctly formatted pandas dataframe is returned from the `run_pipeline()` function. In this example script, we make use of the popular isolation forest (Liu et al. 2008) algorithm to perform our initial anomaly detection.

```python
pipeline_iforest = isolation_forest.IforestAlgorithm(
                                        force_rerun=False, output_dir=output_dir)
anomalies = pipeline_iforest.run(features)
```

One can see that the procedure is the same as always: a PipelineStage object (in this case IforestAlgorithm) is created and then executed on the dataframe using the `run()` function.

**Astronomaly** assumes the anomaly scores are of the type “larger is more anomalous”, which is not the case for the output of isolation forest. Thus we make use of a simple converter to put the scores on a scale of 0 to 5, the same as the user interface.

```python
pipeline_score_converter = human_loop_learning.ScoreConverter(
                                        force_rerun=False, output_dir=output_dir)
anomalies = pipeline_score_converter.run(anomalies)
```

3.6. Active learning

In the galaxy_zoo_example script, there is a small set of code which checks if labelling has already been done. This does not need to be changed generally.

In order to perform active learning, something slightly different to the previous steps must be done. An object, of type PipelineStage, must be created which is capable of executing an active learning algorithm and returning a dataframe with the new scores. **Astronomaly** currently provides one active learning algorithm, as described in Lochner & Bassett (2021), which is created as follows:

```python
pipeline_active_learning = human_loop_learning.NeighbourScore(
                                        force_rerun=False, output_dir=output_dir)
anomalies = pipeline_active_learning.run(anomalies)
```

In **Astronomaly**, we assume that the active learning algorithm can take in a set of features, the original anomaly score (from the machine learning algorithm) and a set of human applied labels, and then return a corresponding dataframe with a new retrained score. The frontend needs access to the actual object, in this case the NeighbourScore object, to be able to run the active learning on demand. This differs from the other stages of the pipeline (such as feature extraction or machine learning) which are static and do not change as the user interacts with **Astronomaly**.
3.7. Visualisation

Finally, we choose to enable visualisation of our feature space using t-SNE (Van der Maaten & Hinton 2008, although any other similar algorithm can be used), which is called in much the same way as the previous steps.

```python
pipeline_tsne = tsne_plot.TSNE_Plot(
    force_rerun=False,
    output_dir=output_dir,
    perplexity=100)
t_plot = pipeline_tsne.run(features)
```

3.8. Return requirements

It is very important that a return statement is included in the `run_pipeline()` function with all the required keys in the dictionary.

```python
return {'dataset': image_dataset,
        'features': features,
        'anomaly_scores': anomalies,
        'visualisation': t_plot,
        'active_learning': pipeline_active_learning}
```

3.9. Running Astronomaly

The script is then passed directly to the `run_server` script as follows:

```bash
python frontend/run_server.py scripts/galaxy_zoo_example.py
```

The pipeline will execute and display some text when it is finished:

```plaintext
##### Astronomaly server now running #####
```

Opening that link with a browser will then display the Astronomaly frontend, similar to Figure 2.

4. The Frontend

After calling `run_server()` and inputting a script, Astronomaly will launch the frontend. This is built using JavaScript, primarily making use of the React\(^6\) framework, allowing easy extendibility. The frontend display is split into two tabs: the anomaly tab and the visualisation tab.

\(^6\)https://reactjs.org/
4.1. Anomaly tab

As can be seen in Figure 2, the anomaly tab allows the user to interact with the data and score objects based on how interesting they are. The data can be ordered several different ways: randomly (to get a sense for the average objects in the dataset), according to the machine learning score and (after active learning is applied) according to the retrained score. The “retrain” button allows the active learning algorithm to be called after labelling some data and the “delete labels” button can be used, with caution, to reset the labels that would otherwise be saved between sessions.

4.2. Visualisation tab

Figure 3 shows the visualisation tab of the Astronomaly interface where a large fraction of the data set can be visualised, in this case using t-SNE (although any method of reducing the data down to two dimensions can be used and is changed in the python script). This tool is helpful to learn patterns in the data and evaluate how effective the features are at not only detecting anomalies, but also grouping similar anomalies together. The type of plot shown on the right will change depending on the data type used (see subsection 4.3 for a discussion on how the plots are generated).

![Screenshot of the visualisation tab of the Astronomaly frontend interface](image)

Figure 3. Screenshot of the visualisation tab of the Astronomaly frontend interface. The features are visualised using some form of dimensionality reduction or manifold embedding, in this case t-SNE. Each point on the plot in the left represents a different object in the dataset. The points are coloured according to anomaly score (lighter is more anomalous). Clicking on a point on the plot will display the data for that object.
4.3. REST API

To connect the Python backend with the JavaScript frontend, we built a REST API using Flask\(^7\). A REST (Representational state transfer) API (application programming interface) is a simple set of rules of communication between two programs, making use of requests and responses.

The key components of Astronomaly’s REST API are found in the frontend package. The interface module implements a Controller class which facilitates calls to the backend. It takes the Python script (such as the one discussed in section 3), executes the pipeline and then has several other functions designed to be called from the frontend.

The run_server module implements the key REST API functionality. It uses Flask to generate requests to the frontend and handle responses. For example, when a user changes the object they are viewing, the frontend generates a request, that contains the index of the new object, which gets passed to get_image() function in the run_server module. That function posts a response, which is the actual image in bytes that can be displayed directly on the web page.

The type of data (images, time series or raw features) is stored as an attribute in the Python Dataset object and read in when Astronomaly is called. The data type will dictate the kinds of plots the frontend will display. This flexibility is implemented in the PlotContainer.js module which will use different components for different data types. This makes it easy to use the same code to work with a variety of astronomical data and to extend the frontend to include other types of plots.

5. Extending Astronomaly

Astronomaly is designed to be modular and flexible, and hence easy to extend. As mentioned before, because Astronomaly relies on a Python script and not on pre-defined controls, one can insert almost any kind of external analysis. However, it is straightforward to incorporate a new element into the Astronomaly library.

Astronomaly makes extensive use of inheritance to allow new classes to inherit a standard set of behaviours such as automated logging and automatically detecting if a pipeline stage has already been run. There are two types of base classes in Astronomaly: the Dataset class, and the PipelineStage class. If the user has a new dataset to be read into Astronomaly which cannot be read by an existing data reader in the data_management package, a new class of type Dataset must be created. If however, the user requires a new feature extractor, machine learning algorithm or essentially any other new step in the pipeline, a new class must be created which inherits from PipelineStage.

Technical details of how to extend Astronomaly can be found in the online documentation: https://astronomaly.readthedocs.io/en/latest/contributing.html

\(^7\)https://flask.palletsprojects.com/en/2.0.x/
6. Conclusions

We have discussed some of the implementation details of the *Astronomaly* package - a framework for personalised anomaly detection in astronomical data. With a modular Python backend, JavaScript frontend and REST API connecting the two, *Astronomaly* is flexible enough to be extended to include new data types, feature extractors, machine learning algorithms and frontend components.

*Astronomaly* is still being actively developed. We plan to implement easy functionality to search a variety of online databases by coordinates to facilitate quick follow-up of interesting sources. We will also include a clustering component for rapid labelling of data, similarity searches and better export of the anomalies found. Finally, development into more effective feature extractors and machine learning algorithms is ongoing, and contributions from the community are more than welcome.

References

Itseez 2015, Open source computer vision library, https://github.com/opencv/opencv

Liu, F. T., Ting, K. M., & Zhou, Z. 2008, in 2008 Eighth IEEE International Conference on Data Mining, 413

Lochner, M., & Bassett, B. A. 2021, Astronomy and Computing, 36, 100481. 2010.11202

pandas development team, T. 2020, pandas-dev/pandas: Pandas. URL https://doi.org/10.5281/zenodo.3509134

Van der Maaten, L., & Hinton, G. 2008, Journal of Machine Learning Research, 9, 85

Wes McKinney 2010, in Proceedings of the 9th Python in Science Conference, edited by Stéfan van der Walt, & Jarrod Millman, 56