Outlier Detection Package Documentation

Norbert Bodendorfer

October 2023

1 Introduction

This document outlines the structure and design of the outlier detection package. Its goal is to convey major design decisions and possibilities to extend the code. It does not contain detailed descriptions of classes, functions etc. Those are contained in the docstrings and are available as a compiled documentation under `INV_anomaly_package/docs/build/html/index.html`. In case the documentation is outdated, it can be rebuild with

```
make html
```

in the docs/ directory.

2 Fundamental design decisions

2.1 Core idea

The current state of the art in outlier detection is characterized by a multitude of existing algorithms, each suitable for a limited set of anomaly types. The available algorithms have a wide range of required computational power and reliability. In particular those requiring the training of neural networks may give unstable results.

It therefore makes most sense to

1. Use and ensemble of algorithms (or "detectors") that is able to detect all of the anomaly types that are present (or we are expecting) in the data we are considering. An anomaly is detected if at least one detector is highly confident about the detection.

2. Restrict the set of detectors to the most efficient and stable subset of all possible detectors achieving the former. This ensures scalability to large amounts of data as well as reliability (not requiring humans to continuously check sensibility of the results).

2.2 Preprocessing and detection

Abstractly, the goal of outlier detection is to identify behavior that deviates from normal behavior. Hence, it is a two step process:

1. We use past data to infer normal behavior, keeping in mind that past behavior may already be polluted by anomalies. We then compare present data to the normal behavior and compute the deviation.

2. Given the deviation from normal data, we infer how likely it is to have encountered an anomaly. This step splits into two sub steps, one of which is the application of several detection algorithms resulting in "outlier scores", the second one being an interpretation of these scores and extracting a yes or no answer.

Since these two steps are independent and may occur via several different algorithms in different detection procedures, they are implemented as independent steps in the code as well.

Step one is implemented as preprocessing of a given time series. An example would be to fit a SARIMA model to a time series and subtract the former from latter. The resulting series contains the deviation of the measured data from a (assumed to be reasonable) model that generates non-anomalous data.

Step two is implemented as the application of detectors to the preprocessed series. An example would be to compute how many standard deviation a measured value is away from the mean (i.e. the fitted model from step 1). Detection of an anomaly would be achieved the number of standard deviations exceeds a certain threshold (so far, 4σ seems to be a reasonable compromise between type I and II errors).

2.3 Train / Test splits?

In many machine learning applications involving predictions, it is necessary to split the data into training, test, and possibly validation sets. It is not directly obvious to which extend such splits are necessary in the case of outlier detection. From a "prediction" point of view, one may argue that we observe past data (a subset of all data), build our model (preprocessing) on that data, and then predict into the future (to present data) to identify outliers via comparison.

For outlier detection, the "prediction into the future" step is however not logically needed. Rather, one asks the question whether there exists a reasonable model for the data that describes already measured values well (as opposed to trying to predict values that have not been measured yet). In fact, the underlying model may be time dependent (e.g. have repeated trend changes) that are not directly linked to anomalies.

In the code, this is currently implemented as follows.

- Preprocessing is performed on the complete time series. This makes the task technically simpler (extracting the model as e.g. `model.fittedvalues`)

and more stable in case of fitting ARIMA models. (There is no need to evaluate model performance on a validation set)

- Detection is split into training and test sets. Detectors are calibrated on the training set and evaluated on the test set. It is possible to set the whole series as the training set.

3 Code structure

There are two ways to access the outlier detection mechanism.

- Through the class `UnivariateOutlierDetection`, which contains methods for automatic detector selection, preprocessing, and detection. Details are outlined below.
- Through "one-line-calls", e.g. `process_last_point(ts, ts_dates)`, where the arguments are lists of the time series values and measurement timestamps. These functions instantiate a `UnivariateOutlierDetection` object and call all needed member functions to obtain outlier scores. An example call of a one-line-function is in `web_api/flask_app.py` . There are several additional arguments that can be passed to the one line calls to fine tune the detectors. These are listed in the html documentation.

The typical workflow of the outlier detection mechanism is as follows, see the one-line-calls in `univariate.py` for details:

1. The class constructor is called with the time series as argument. Several sanity checks on the data are performed.
2. The `AutomaticallySelectDetectors()` member function checks the data for seasonality and models it with ARIMA models. In case "sufficiently good" model is found (judged via R^2 scores), seasonality and ARIMA models are subtracted. Seasonality is tested only for a reasonable set of seasonalities (hourly, daily, weekly, yearly).
3. The `WindowOutlierScore()` member function is called and return raw outlier scores for a selected test window.
4. The `InterpretPointScore()` member function is called with the scores of a chosen timestamp to yield a json object containing a detecting report, including a description of how an anomaly was identified.

Comments

- Testing for arbitrary seasonalities is expensive and unreliable. Unless we know that non-standard seasonalities will occur in our data, we should not try to model them.

- Fitting arbitrary models to the time series is expensive and prone to overfitting. So far, SARIMA models gave good enough results for the available data (OSKAR, INVTEREST, Numenta (up to very special anomalies), synthetic SARIMA + anomalies).
- The code treats seasonality and ARIMA separate. Seasonality is implemented via `seasonal_decompose` from statsmodels, either additive or multiplicative (which is better). ARIMA is implemented also via statsmodels. The code also has the possibility to compute a joint season-trend decomposition via statsmodels `MSTL`, but this gave unsatisfactory results in tests, resulting in spurious anomalies due to poor trend estimations. Hence, `MSTL` is not used in `AutomaticallySelectDetectors()`.
- The `UnivariateOutlierDetection` also allows for manual fine-tuning of detectors and preprocessing. Various preprocessors are implemented in the file `univariate_preprocessors.py`.

4 Detection of anomaly types in existing data

Detection of the different types of anomalies are outlined in several notebooks in the `INV_outlier_detection` repository (other than `INV_outlier_package` containing the current version of the code). `INV_outlier_detection` contains an outdated version of the outlier detection algorithm and I left it there for now to be able to directly test changes in `INV_outlier_package` as follows:

1. Clone `INV_outlier_package` and install the package with poetry install instead of pip (still do `pip install -r requirements.txt`).
2. Clone the `INV_outlier_detection` package containing the notebooks and set up as in the Readme. (virtual environment, `pip install -r requirements.txt, poetry install`). Then, manually change the link to the package in `venv_p3.10/lib/python3.10/site-packages/outlierdetection.pth` to that of the package above. Then, the detection notebooks (running in their own virtual environment) always use the most recent code from `INV_outlier_package`.

The executed notebooks including results on the analysis are available in html format in the folder `notebooks_html`. Some of these html files do not contain all of the available time series to keep their size managable.

5 Multivariate data

Multivariate anomalies can be detected via the `MultivariateOutlierDetection` class. It functions similarly to `UnivariateOutlierDetection`, but has not been tested in as much detail. So far, the same preprocessing is applied to all series.

Available detectors are Mahalanobis distance and isolation forest. They are very fast and reliable and have the ability to detect correlation outliers.

ToDo:

- Rethink preprocessing: separate for all series?
- Add seasonality according to previous bullet point.
- Documentation once preprocessor logic is clear.
- A generalization of PRE, or a multivariate version of the KS Test to determine in the probability distribution changes seems hard to implement, see e.g.
  `https://stats.stackexchange.com/questions/71036/test-if-multidimensional-distributions-`

6 Planned maintenance

6.1 Functional update routine

The main repeated functional update step for the code is as follows:

1. Apply the detection to newly available data and check whether the expected anomalies are detected.
2. If the detection is unsatisfactory, understand why detection fails (both for type I and II errors) and add or fine-tune some of the detectors or the preprocessing.

6.2 Planned updates and refactoring

- Tree-like preprocessing:
  A possibility to improve the preprocessing would be to stack several preprocessing steps in a tree-like structure, starting with the original series and creating new leaves with the several preprocessing steps available at that level. Then, we perform detection at the lowest leaves on a branch. This way, intermediate preprocessing results can be reused, saving computational power.
- The preprocessors should be derived from a baseclass that fixes the input / output interface. Docstrings should also be inherited.
- All possible points where mathematical operations such as inversions can fail should be put into try / catch statements, e.g. also detectors.
- Improve handling of colinearity in MD detector. For now, np.nan is returned as the covariance matrix is not invertible.

6.3 Possible update ideas:

- Convolve the time series with signatures of discrete n-th derivatives after differentiating. This allows to detect at which order of differentiation an outlier was a spike, since, a spike (Dirac detla like) looks like a quick up down succession (derivative of Dirac delta) after differentiation.
- Increase series preprocessing before modelling, e.g. via Box-Cox transformation.
- Add functionality to detect increases in data correlation, e.g. two uncorrelated time series become suddenly correlated. The currently implemented detectors won't detect this, while they already do detect the other way around (loss of correlation via increased Mahalanobis distance).
- Increase the number of mathematical models fit to the time series if necessary.

7 Tests

Unit tests are available via pytest. In the folder outlierdetection, run `pytest`.

8 Profiling

The unit tests can be profiled as follows:

1. Run
   `python3 -m cProfile -o profile -m pytest tests/`
   in the outlierdetection folder. This generates the file `profile`.
2. Run
   `gprof2dot -f pstats profile -o callingGraph.dot`
   to turn `profile` into a graph called `callingGraph.dot`.
3. Inspect with an .dot file viewer, such as the online tool
   `https://dreampuf.github.io/GraphvizOnline`
   by using the content of the `.dot` file as input. If needed, export as an svg and view with another viewer, e.g. firefox.

9 References

A Workflows

A.1 github flow

Creating a new feature and merging to main:

1. Create new branch and switch to it:
   `git checkout -b <new_branch_name_develop>`
2. Make changes (`git add, commit, push`)
3. Get up to date stuff from other peoples changes
   `git fetch`
4. Bring the latest commits of master to your branch
   `git rebase origin/main`
5. Switch to the master branch
   `git checkout main`
6. Get the latest changes from master
   `git pull origin main`
7. Merge the changes
   `git merge <new_branch_name_develop>`
8. Pushing local changes to the remote repository
   `git push origin main`

A.2 peotry version bump

Version control and package building is done via poetry.

1. `poetry version patch`
   `>>> Bumping version from 0.1.0 to 0.1.1`
2. `poetry build`
3. `git add outlierdetection/dist/*`
4. `git commit -m "new version patch"`
5. `git push --tags`
6. `git tag 0.1.1`