Anomalies Detection

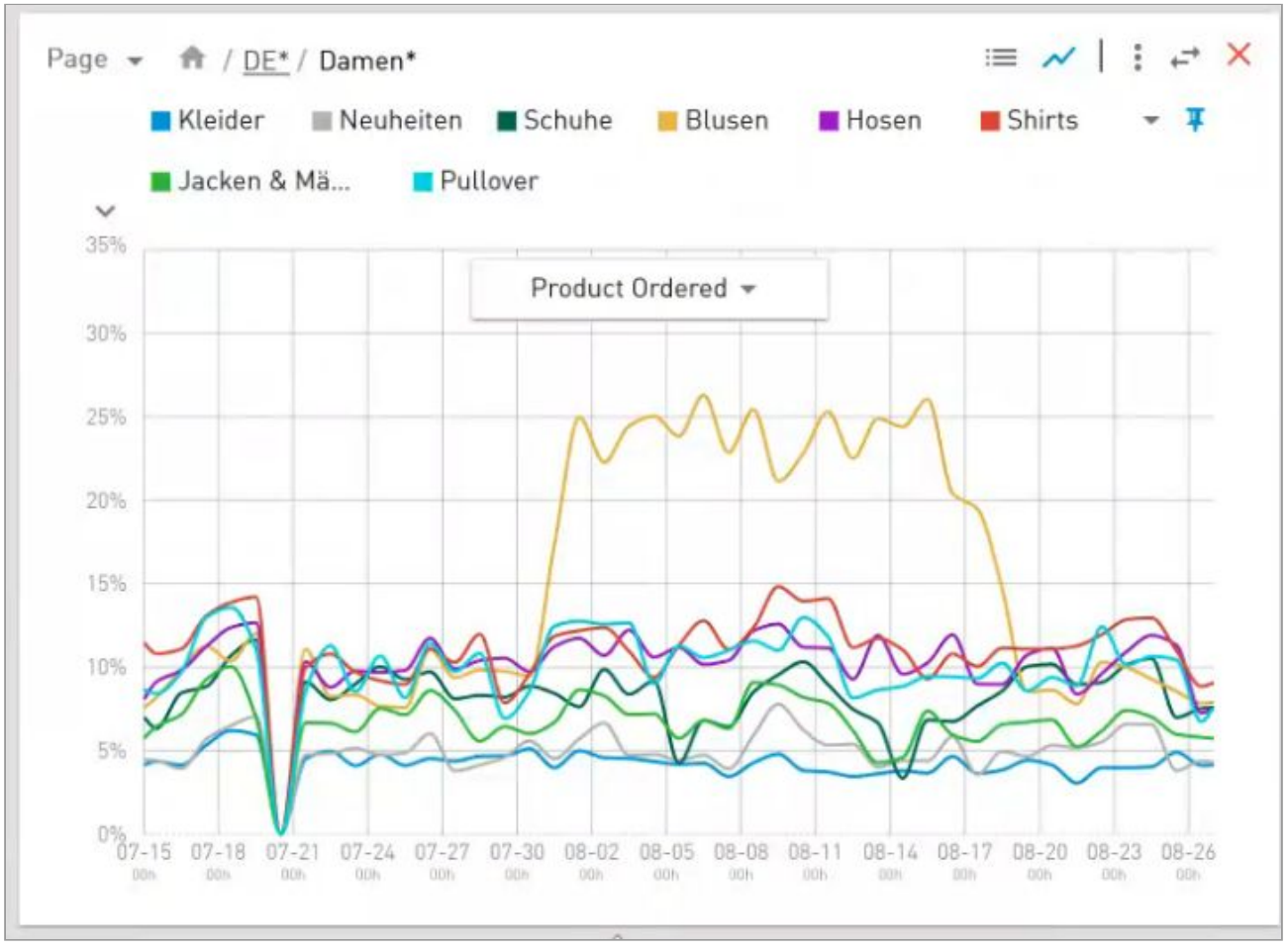
Overview

Main goal of the project is to create a way to **automatically** detect possible problems occurring during everyday work of our customers.
Quite often we learn about an issue that occurred on the customer's site weeks or even months late. In each of those cases investigative analysis shows that the problem could have been detected by one glance at a correct chart. This task can be done as well by a pretty simple algorithm.

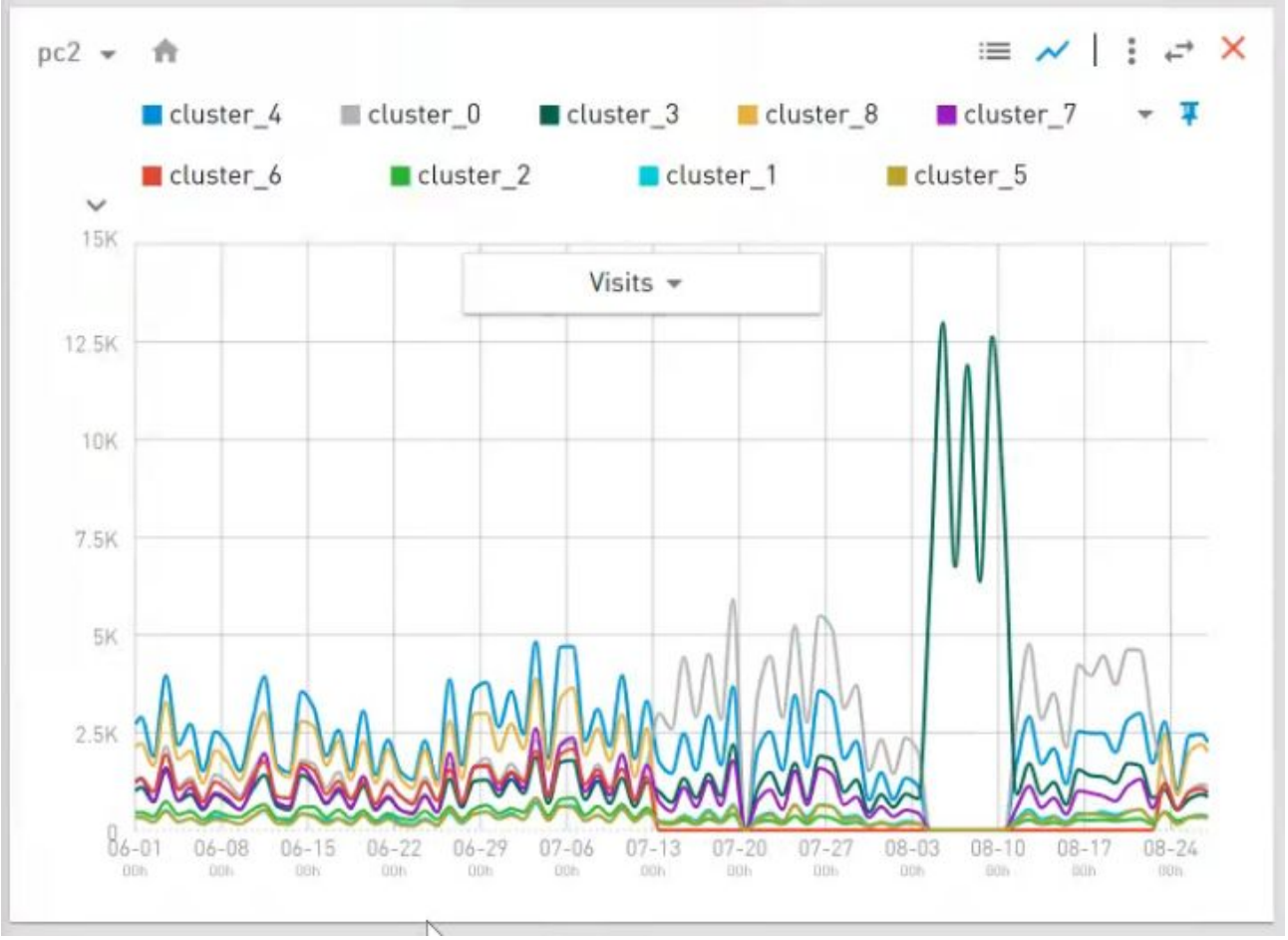
Problems

Range of possible problems we may detect is quite wide. If we automate analysis of every attribute on site by every given metric and will look for strong outliers in the data we can catch almost any anomaly.

One example is a bot that was used by the site developers in order to test the checkout system. They did not disable tracking systems for this bot's activity… It has "ordered" so many baskets containing the same blouse that an average basket fell through the floor and alerted almost everyone until the puzzle was solved :).
The average basket amount anomaly was detected days after the anomaly started.
Outliers from this chart would have alerted everyone right away that something is off.



Another example is a failure on a third-party data provider side. For a week the price clusters information for the site users was not provided correctly. As a result almost all the users got into a single cluster and were treated incorrectly. Again, quite easy to see the outliers when the correct chart is presented.

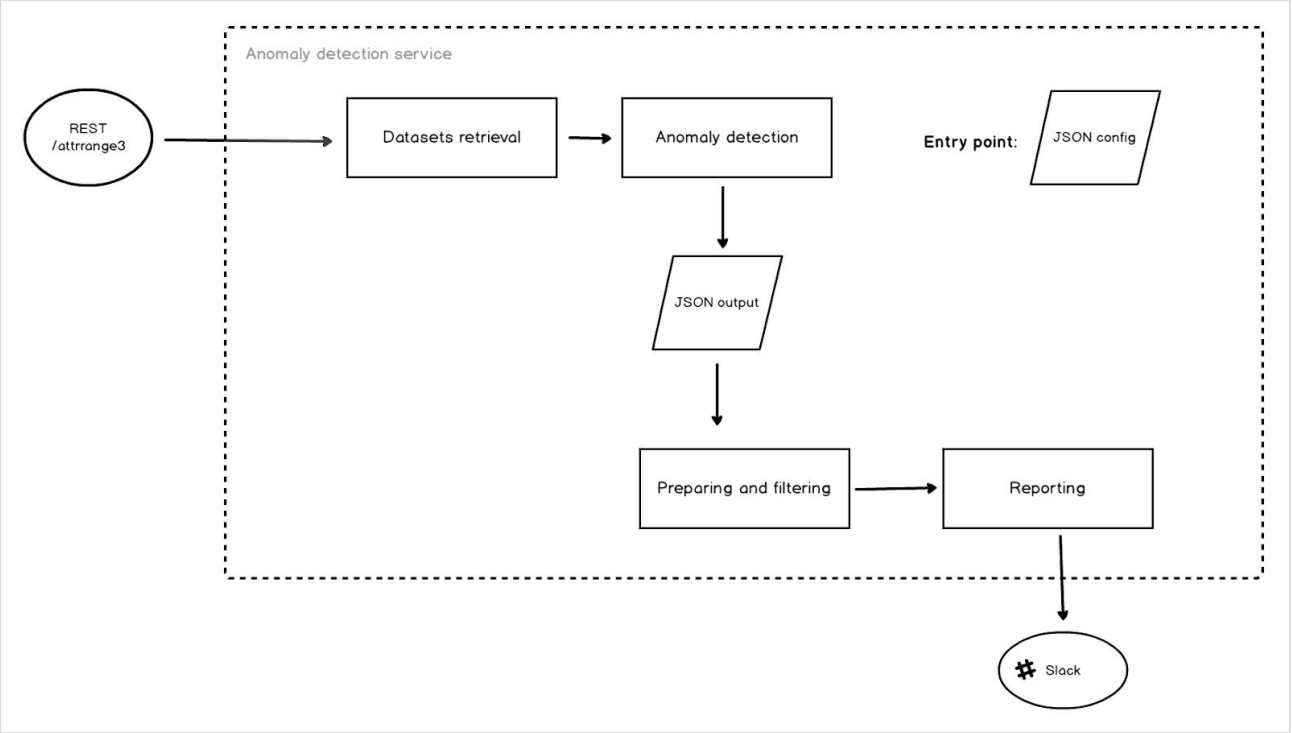


Suggested solution

As demonstrated, there is always a dataset which contains outliers indicating the problem. The solution can be split into several steps:

1. Automate gathering of such datasets (exhaustive amount to cover all possible cases).
2. Automate outliers detection onto any given dataset.
3. Combine parts 1 and 2 together and teach them to exchange data.
4. Provide a mechanism for “accepting/approving” anomalies in order to prevent reporting the same alarms over and over.
5. Provide on top a reporting system to be able to react to the detected anomalies.
6. Automate the whole process so it is repeated at regular intervals and for all siteids.

Below is a basic chart and then each step is discussed in more details



Few ground rules:

- We assume that the tracking data collected for similar consecutive time periods is presenting a sample with normal distribution (even though it may not always be strictly the case).
- We use a well known rule of 30 being the minimum sample size allowing us to apply certain statistical rules to samples.

1. **Automate generation of relevant datasets**

a) There are 2 very important parameters for getting the datasets - length of the period we take into account and a time step.
The main problem here is to find a good balance of accuracy and relevance. Since we get updates of the tracking data daily, **one day** is the first time step that comes to mind. So the length of the period of 30 days sounds about right as well.
Taking days as a step is suffering from mixing together weekdays, weekends and even some holidays (e.g. Christmas), but looks like a good balance anyways.
For example, taking an hour as a step sounds a bit extreme as we will have too big diversity of the visits (night vs day) so the model will not be very accurate and also learning that one or another hour was an outlier may produce not very relevant info (too many false positives warnings and errors).
However, let's keep an opened mind here and keep both of these parameters as a part of the service configuration:

```
{
  Datasets: [ {
    siteId: "brax",
    TimeAgo: "30d",  // 3d, 7d ...
    TimeStep: "1d",   // 1h, 3h ...
    OutliersDetectionMethod: "3-sigmas"  // "last-point"...
    MetricesList: "Revenue", "Basket" // "all", "Visits", "NcYzTUmBXTYhQQgQ0ji26g"...
  },
```

{...}, ...]
}

b) Next important thing is to check **all** the possible combinations of attributes/attribute values and metrics. A call like https://beta.odoscope.cloud/felgenshop/attributes will provide an entry point for a loop which then should recursively parse the whole attribute values tree while gathering the datasets by every possible metric. This should provide thousands of collected datasets. E.g. for felgenshop we have 34 “root” attributes (with trees of values) and 9 metrics.

c) Limiting attributes/values list.

To prevent a total calculations explosion I suggest to limit the amount of attribute values taken into consideration. Current proposition:

```
MinVisitorsPerTimeStep: 30   // 100 etc.
```

The algorithm multiplies this number by the amount of Time Steps. E.g. for 30 days and 1 day Time Step we will get the number of 30 * 30 = 900.

This number is used as a filter while choosing attribute values to process. If total visits related to this attribute value is lower then the number calculated above we skip this attribute.

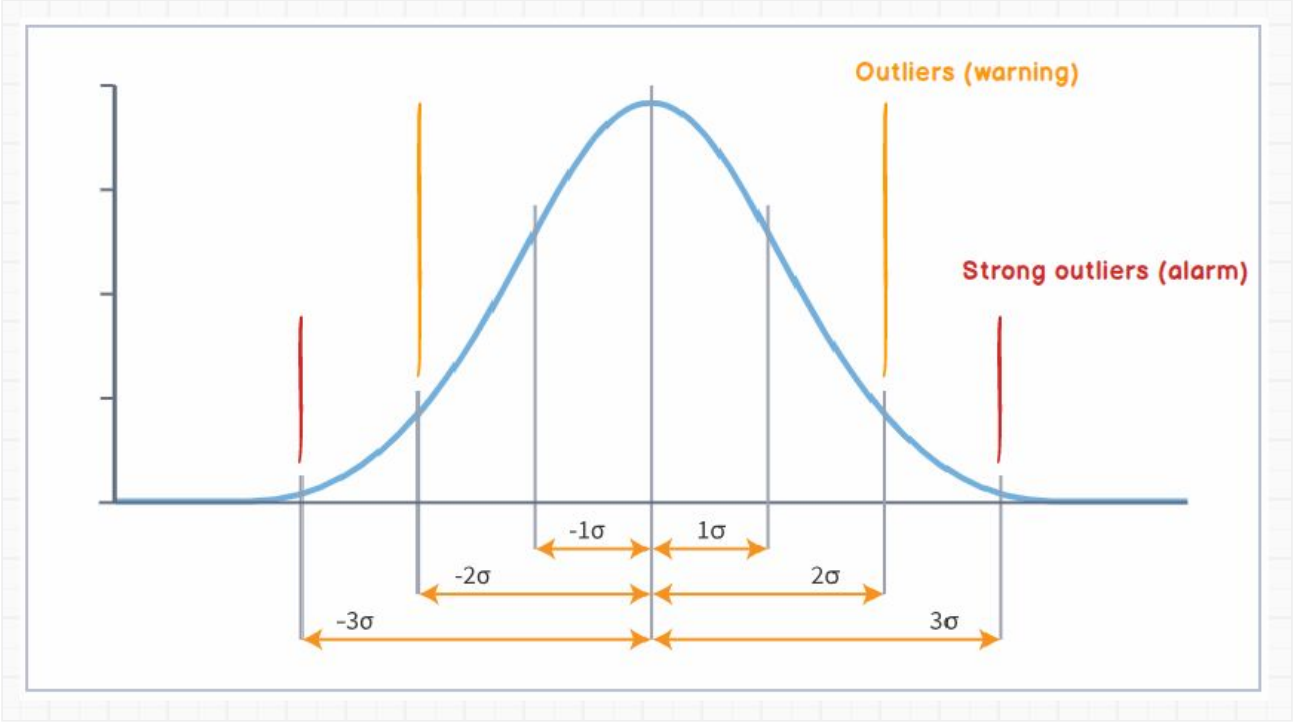
On top of that we apply a way to manually configure the depth of scan steps and limit the number of attribute values to be taken into account on a per attribute basis.

E.g. a way to say “Take only 1 level for browsers (ignore the versioning of each browser)” or “take 2 levels of pages and take only top 5 attribute values there”.

2. **Automate outliers detection onto any given dataset**

This task should be solved without connection to a way of gathering the data, so the method could be easily debugged, replaced by another one or several methods at once could be used (in the future).

The method called “3-sigmas” is described below. More methods coming soon.



Using the given dataset an algorithm calculates the **mean** value and the σ (standard deviation). After that any point of the dataset lying within 2σ from the mean is considered an outlier (reported as a warning) and any point of the dataset lying within 3σ from the mean is considered a strong outlier (reported as an alarm).

Values 2 and 3 here should be parameters of the algorithm:

```
{
...
OutliersDetection: {
   OutliersMultiplier: 2  // 1.8, 2.5 etc.
   StrongOutliersMultiplier: 3   // 2.5, 4 etc.
  }
}
```

3. **Combine parts 1 and 2 together and teach them to exchange data.**

Just a system that feeds data from system 1 to the system 2 and produces an intermediate JSON file with the results. And example of such JSON:

```
{
siteId: “brax”,
OutliersDetectionMethod: “3-sigmas”,
checkTimeStart: “2020-09-07T03:00:01+00:00”,
checkTimeEnd: “2020-09-07T03:05:32+00:00”, // so we know how long it took to process the data
TimeAgo: 30d,
TimeStep: 1d,
DateStart: “2020-08-07T00:00:01+00:00”
DateEnd: “2020-09-07T00:00:01+00:00”
Result: {
  Warnings: [],
  Alarms: [
     {
       OutlierPeriodStart: “2020-09-05T14:00:01+00:00”,
       OutlierPeriodEnd: “2020-09-06T14:00:01+00:00”,
       Metric: “Visits(qGaHiHepUD1pLl_cWoD9aQ)”,
       Attribute: “Device Type > Tablet > 10”(eAeciHepUD1pLl_cWoD932)”, // full path so it’s easy to find
      }
    ]
  }
}
```

4. **Provide a mechanism for “accepting/approving” anomalies in order to prevent reporting the same alarms over and over**

This one is not yet figured out. Will update this part closer to the we’ll need it… but it is clear that a system like this will be needed. And most likely it will work with the “raw” JSON provided after the anomalies are detected.

5. **Provide on top a reporting system to be able to react to the detected anomalies.**

For now we need a pretty crude system that is scanning recently made JSON files and reporting detected Warnings and Alarms to a relevant slack channel. Later this module will be responsible for providing API output but for now let’s do some dirty reporting.

6. **Automate the whole process so it is repeated at regular intervals and for all siteids.**

Pretty self-descriptive title :) We need to start with a single siteid, catch all possible bugs and edge cases first. And then spread it to all other sites…