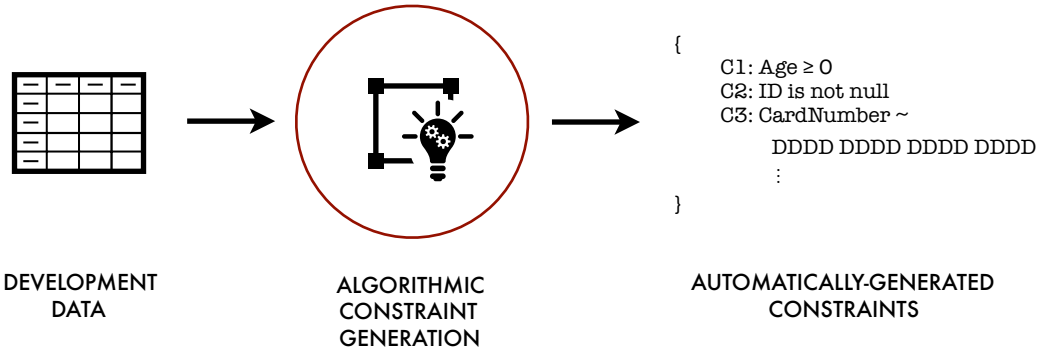
AUTOMATIC CONSTRAINT GENERATION AND VERIFICATION

Test-Driven Data Analysis Series • Stochastic Solutions Limited

Correctness is a key problem at every stage of data science projects: completing an entire analysis without a serious error at some stage is surprisingly hard. Even errors that reverse or completely invalidate the analysis can be hard to detect. *Test-Driven Data Analysis* (TDDA) attempts to identify, reduce, and aid correction of such errors. A core tool that we use in TDDA is *Automatic Constraint Discovery and Verification*, the focus of this paper.

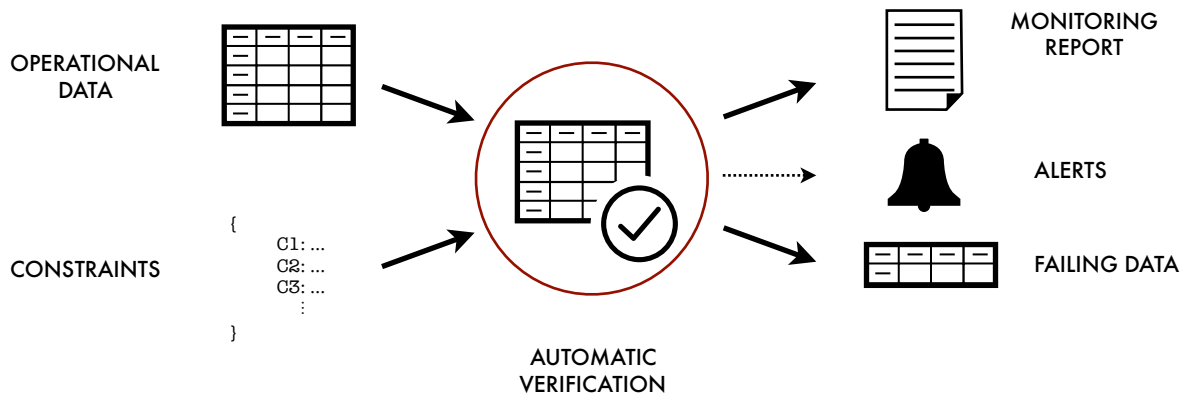
Automatic Constraint Generation

Constraints constitute a powerful mechanism for precisely describing properties we expect of data. There are typically many things we know should be true for any source of data and in principle we could write these down and use them as consistency checks for our processing. This is true for intermediates and output datasets as well as input datasets. In practice, manually determining and capturing constraints is time consuming and is likely to result in rather patchy coverage. The key breakthrough with TDDA's constraint discovery process is that constraints are produced *algorithmically* from the data.

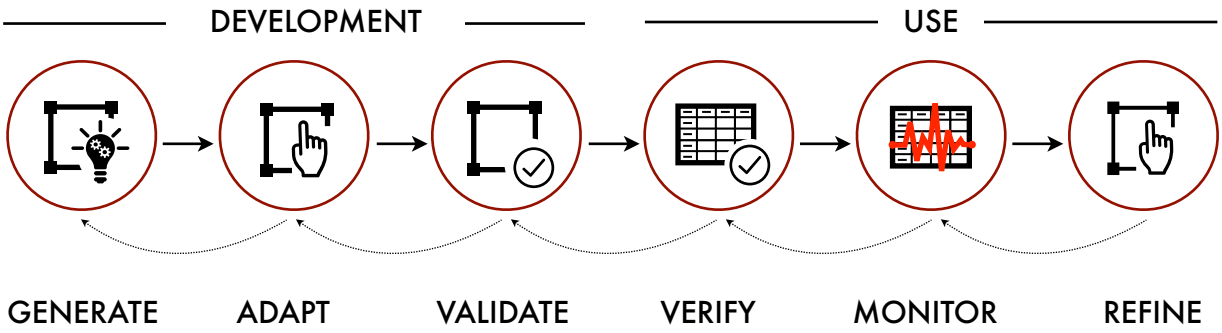


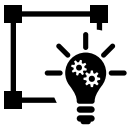
Data Verification with Constraints

Given a set of constraints for each dataset, we can use a verification process to check that everything is as it should be at each stage of our processing or analysis. Ideally, verification should be carried out automatically and monitored to detect problems as they arise, feeding to a mechanism for investigation and correction or other handling.

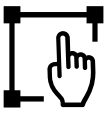


The Constraint Generation and Verification Cycle

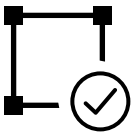




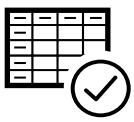
GENERATION. As described above, the first stage is normally to run the generation process on one or more initial datasets. If these are believed to contain bad data items, we call this constraint *inference;* otherwise, we call it constraint *discovery*. Although the resulting constraints can be used immediately to verify other data, where possible the practitioner should first examine the constraints produced to see whether they contain anything unexpected or—just as importantly—fail to include something that was expected. If, for example, the suggested minimum and maximum ages (for humans) produced by discovery were, respectively, –38 and 17,031, this would be a clear indication that there were invalid ages in the generation data, which should probably be corrected or excluded. Conversely, if the constraint generation process did *not* result in a constraint saying a CustomerID must be non-null, this would indicate that there were records with no CustomerID in the development data, which might well be a serious problem. When constraint inference is used, the constraints might prohibit valid data, in which case they should be relaxed. Ordinarily, the results of constraint generation are written to a JSON file, which can be read by the verification process and used to verify subsequent datasets.



ADAPTATION. As a result of looking at the output, the practitioner will usually decide to change the values of some constraints, to remove others, and to add still others that were not satisfied by the data, but which should have been. There are two main ways to do this: one is simply to edit the constraints by hand; the other is to remove (or correct) the bad data that caused suboptimal constraints to be produced.[1] Often a combination of both will be used. A third approach is not to bother with this stage in the first place and simply to handle failing constraints as they occur. Such an approach can be valid, but is not really capable of handling situations in which a desirable constraint was not discovered as a result of problems with the initial data used. To that extent, failure to examine the outputs of discovery can dilute the potency of the approach.



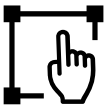
VALIDATION. The process of adapting constraints is part of a larger process of *validating* the constraints. In many cases a further level of validation is carried out by verifying other data using the same constraints, and checking that the system of constraints is both catching bad data and letting good data through.



VERIFICATION. Once a set of constraints has been accepted, the idea is to use them to verify newly arriving or newly produced data. Automating the verification will tend to reduce the scope for omission, variation, and human error. Depending on the nature of the data being used, in some cases a new dataset being verified will overlap with data previously verified. In these case, it may be desirable to prevent records or values that have previously been flagged as bad from being reported again. We have developed a system for tracking records that have previously failed constraints, with the option of suppressing warnings about those in the future. This includes controls over how to handle records that are bad once, then get corrected, then go bad again.

[1] In some cases, it would also be possible to add/modify data specifically to cause a constraint to get generated with a particular value, but this would be more unusual and has some obvious dangers.

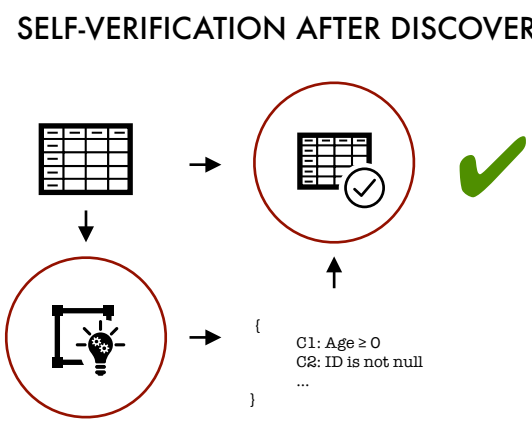
MONITORING & ALERTING. If verification is automated, there needs to be some kind of monitoring and alerting mechanism for data that fails constraints. Internally, we use a mixture of systems for this, depending on the style, granularity, and nature of the data feeds and analytical processes. Often, different constraints have different levels of importance, so it may be that a range of levels of alerting is appropriate.



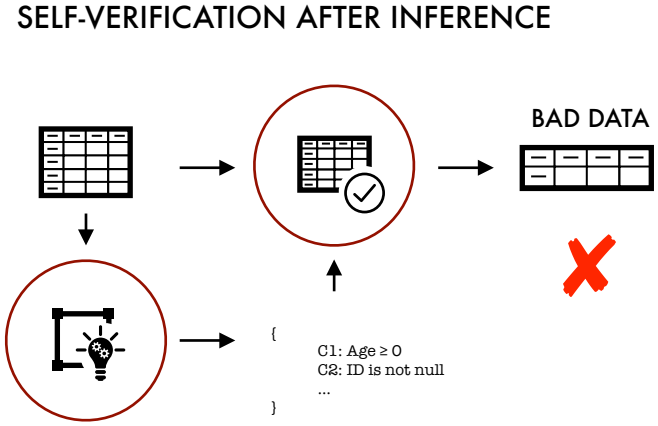
CONSTRAINT REFINEMENT. In a typical deployment, it will be useful to refine the constraints over time as part of monitoring and responding to failures. As more verifications are performed, it is often necessary to relax some constraints, as a broader range of data is processed. For example, previously unseen, but valid categories may appear, as may valid numeric values that the constraints classify as "out-of-range". Ideally, however, over time data and processes improve, so that constraints that would not have been useful on the data as it was during the initial discovery phase now become workable. (For example, perhaps there were missing values in the early days, but data capture has improved to the point that these are now rare or non-existent.) For this reason, it is useful to re-run the discovery process periodically on more recent data to see whether new constraints are viable. In this way, rather than constraints only ever being relaxed, they can sometimes be tightened, or new constraints can be added. We are also considering supporting constraints that are true only over subsets of the data (e.g. apply this constraint only to data after some date, or to all data except records from a particular site), which is a potentially powerful extension.

Generation from Good Data vs. Generation from Bad Data: Discovery vs. Inference

In its simplest form, the algorithmic constraint generation process in TDDA assumes that the example datasets provided to it represent "good" data. We refer to this as constraint *discovery*, as the system works to characterise the assumed good data. When used in this mode, the only constraints generated will be those that are completely satisfied by the development data used, and verification of that development data with the discovered constraints is guaranteed to succeed.[2] When run on a different dataset, of course, some constraints may fail, and it is detecting such situations that is our primary goal.



When we do not assume that the development data represent only good data, we call the constraint generation process constraint *inference*. This process is more powerful but obviously occasionally generates inappropriate constraints, as the bad data values are not provided to the algorithm. After constraint inference has been performed, in general a verification over the development data will result in some records and values being marked as bad (i.e., failing the constraints). As a result, constraint inference can not only find useful constraints for verifying new data, but can also find potentially bad data without any explicit guidance.



2 This is broadly true. In fact, if time/date-based fields are used, and constraints are generated on how far in the future/past they are allowed to be, they may fail if run at a later time. However, an "as at" parameter can be provided to specify that they verification should be perform *as at* a nominated time, in which case, such constraints should also pass.

When constraints fail, there are several options for how much detail is reported, including producing summary statistics about which constraints failed and how often, which individual values in which records failed, and which records and fields contained any failures. Importantly, the JSON file format used by TDDA is shared across a number of different implementations, currently including Pandas DataFrames, CSV Files, relational databases (currently PostgreSQL, MySQL, and SQLite), document collections in MongoDB, and datasets from our own Miró software. Thus constraints can be discovered on data in one format and used to verify data in another.

Getting the Most Out of Constraints

In addition to using constraints for validation in production processes, and to check data feeds, we find the constraint generation process to be useful from the first stages of data exploration. The constraints reveal a wealth of information about the shape of data, possible problems with its health (either from source, or resulting from imperfect encoding-transmission and decoding). It is also valuable to realise that it is not only input data that the process should be applied to, but also results datasets and intermediate datasets. This is useful not only because the output from one stage of processing is usually the input to another, but also because constraints can usefully identify errors in our own processing.

We can leverage the constraints framework further by creating new fields in our data or new summary datasets that encapsulate properties we know should be true of the data but which are not amenable (at present) to automatic discovery. The kind of constraints that are generated automatically, and which can be represented in the constraints framework, are shown in the box.

KINDS OF CONSTRAINTS

SINGLE FIELD

TYPE:	Age is integer
MIN & MAX:	0 ≤ Age ≤ 120
NULLS:	No nulls in CID
DUPLICATES:	No duplicates in CID
ALLOWED VALUES:	Insured in {"Y", "N"}
PATTERNS:	UUID ~ [0-9a-f\-]{20}

FIELD PAIRS

SEQUENCING:	StartDate < EndDate

FOREIGN KEY RELATIONS

Every value in Orders.PartID must also exist in the field PartID in table MasterPartsList .

Examples of further constraints we might want to force to be included through suitable definition of new columns or aggregates might include:

- A set of percentage columns that should sum to 100% (allowing for rounding errors)
- **if-then** consistency checks such as **if** *day = "Sunday",* **then** *StartTime must be between 14:00 and 17:00.*
- Aggregating transaction totals by day and asserting that the total should not be zero for any day, or perhaps for any branch for any day.
- Splitting a structured string into parts and allowing constraint generation to operate on each part.

Examples of Constraint Verification in Action

SURVEY DUPLICATION

An online service allocated a "unique" identifier to each record in a data feed comprising multiple sources. Occasionally, the same ID appeared more than once in the feed, sometimes indicating duplicate records (which is bad) and sometimes attached to different records (which is worse). TDDA identified these issues, and led to a redesign of the allocation process which ultimately eliminated the duplicate records and IDs, and thus avoided double counting and collisions.

BACKWARDS IN TIME

A website collected surveys after visitors had booked through a partner site. The surveys included the time between the leaving for the partner site and returning. Sometimes this time was negative! Investigation showed several causes, including synchronisation issues, inconsistent timezone handling and some bot activity. TDDA identified these out-of-sequence timestamps, allowing diagnosis and rectification/mitigation of the issues.

THE WHOLE AEROPLANE?

What is "plausible" cost for air tickets? While a typical flight might be in the £50-£2,000 range, a booking for multiple passengers in first class can potentially be several tens of thousands of pounds. But not £450,000,000.
Constraint inference was able to identify highly outlying fares, which in many cases resulted from currency conversion errors, leading to an increase in the fidelity of data gathered.

STRUCTURED STRINGS

A data feed included string identifiers with substructure separated by dashes. Unfortunately, sometimes, the components themselves included dashes, which made splitting out the components ambiguous and error-prone. Constraint discovery generated one regular expression pattern that was matched by all the "bad" examples, and several others for the "bad" identifiers, allowing the bad examples to be fixed.

SAVED BY THE METADATA

Even the humble file includes various sources of metadata—filename, location, timestamps, size etc. In some cases, we know what the relationship between that metadata and the data in the file should be. TDDA can be used to verify the consistency between the data and metadata. We often take advantage of this when working with flat files or JSON files from web services like Amazon S3, especially in the fairly common case in which further metadata is encoded in the filename.

NULL-LAND & NAMIBIA

There is an area of sea in the Gulf of Guinea whose coordinates are 0°N, 0°W. Although there is no land there, a remarkable amount of latitude-longitude data geolocates to this area, which is affectionately known as "Null-land". This usually results from mapping nulls (missing values) to zero. Similarly, when two-letter country codes are used, Namibia is often over-represented, because NA used by systems like Pandas as a null marker. TDDA can help spot both of these problems when they arise.

WHEN DID THAT HAPPEN?

There is perhaps no source of data as prone to errors as dates. Does a timestamp represent time in the Cloud-based data centre, time for the user (who could be based anywhere), or something else?Does it include a daylight savings adjustment? If it does, is that DST at the time of recording, at the time of the event, or as of now? TDDA cannot necessarily answer these questions, but can often highlight when and where there are inconsistencies that need to be addressed.

SPIKES IN THE DATA

Often there are genuine spikes in data, reflecting common prices, popular items, or majority choices.
But when there are spikes at "special" values, extra attention is warranted. 0, 99, –1, 65535, empty strings, cities called "NULL" / "NA", and midnight at the start of 1st January 1970 are all examples of this.
In some cases, TDDA can identify suspicious spikes automatically, and it can always be instructed to flag them.

DATA TRANSFER

A standard way of checking that data has been transmitted accurately is with checksums or hashes, but this really only checks that the ***transmission*** *was OK. If constraints are verified (and perhaps even generated) in source and receiving systems, we can have more confidence that meaning has been preserved.*