Cleansing, processing, and visualizing a data set, Part 1: Working with messy data

M. Tim Jones

Decide common problems associated with cleansing data for validation and processing, with solutions for dealing with them. You'll also find a custom tool to make the process of cleansing data and merging data sets for analysis.

Data comes in many forms, all of it messy. Whether we're talking about missing data, unstructured data, or data that lacks regular structure, you need methods to cleanse data before you can process it to improve its quality. This tutorial explores some of the key problems of working with real-world data and the methods you can apply.

This is the first installment in a three-part series that explores the problem of messy data and the methods available to increase the quality of your data sets (see Figure 1). Parts 2 and 3 will continue with data analysis using machine learning and data visualization.

**Figure 1. Data processing pipeline, from cleansing to learning and visualization**

Data science and its algorithms are clean and precise, but the data on which they operate come from the real world, which means that they're messy and require some preparation before you can use them effectively. The quality of insights you derive from data depends on the validity of that data, so some preparation is required.

Data cleansing has a long history in databases and is a key step in what's known as extract, transform, load (ETL). ETL is commonly used in data warehouses, where data is extracted from one or more sources; transformed into its proper format and structure, including cleansing of the data; and finally loaded into a final target location, such as a single database or file.

Let's begin with a discussion of what it means to have messy data, then dig into some of the methods for deal with that mess.
What is messy data?

Data sets large and small are rarely ready to use. As Figure 2 shows, my simple comma-separated value (CSV) data set has a variety of issues, including invalid fields, missing and additional values, and other issues that will be problematic when I attempt to parse this file for use by a machine learning algorithm.

Figure 2. A simple messy data set example

This example is a simple one, but anyone who has worked with a public data set will understand these issues and the need to preprocess data to make it useful. Data sets that have such obvious errors make the results of the processed data somewhat questionable. The observations with errors result in incomplete data or invalid observations that can lead to incorrect results. Cleansing data is therefore a key step in the data processing pipeline.

Data may also come from multiple sources. Although each source may be valid in isolation, bringing the data together may require processing for consistency and uniformity. For example, one data set may have a different unit of measure for a given field than another, requiring that they be normalized.

One key factor for data validity, then, is the format in which the data is represented. Let's look at that next.

Data formats and schemas

Data sets can be in many forms, but the majority are stored as delimited text files. As shown in prior examples, these data sets delimit their fields by using a character, commonly a comma, but in other cases through white space (space, tab, etc.). These raw data sets are particularly prone to error because they lack any information that indicates their structure and so require data scientists to interpret the data set manually.
So-called "self-describing formats" can greatly improve our ability to maintain data correctly. These formats include XML and JSON. These data formats allow the data to be embedded within metadata to make it fully self-describing within a single file. They also permit complex data formats that are more difficult to describe with simple flat text files (such as variant arrays of data or relationships within the data).

Figure 3 shows how you can represent temperature data by using the JSON format. As shown, the data is labeled, and the labels are predefined such that the ingest tool understands what to expect.

**Figure 3. Using JSON format to self-describe a data set**

```json
[
  required: [Sensor, Temperature, DateTime],
  properties: {
    Sensor: { type: string },
    Temperature: { type: real },
    DateTime: { type: string }
  }
]
[
  Sensor : A1,
  Temperature : 45.2,
  DateTime : 2015-11-25T19:17:53
],
[
  Sensor : A2,
  Temperature : 47.9,
  DateTime : 2015-11-25T19:06"
]
]
```

The downside of using self-describing formats is obvious from the example. In a simple flat text file, the size of the data is small compared to a self-describing format. That said, binary formats do exist that can achieve self-description with much less overhead, but the simplicity of CSV or other flat-file text formats prevail because of their simplicity of parsing. You can define schemas for flat-file text data sets, as I show in the sample application.

**Data blending or fusion**

*Data blending* is the process by which a data set is constructed from two or more independent data sets. Blending data may not be a one-time process; instead, it can be performed on demand based on the machine learning use case.

Blending data has all the problems discussed thus far — namely, the need to cleanse more than one data source. Fusing multiple data sets has additional problems, however, in the representation of the data in each source (such as one data set that uses Celsius and another that uses Kelvin, as illustrated below). The data may not be consistent across sources and may require transforming and reordering data fields so that the fused data can be properly used.
Methods for data cleansing

Data cleansing begins with data parsing, which means taking each observation from its data file and extracting each independent element. You can easily identify parsing if the records are similar (same number of elements, similar types, etc.).

Given the definition of a schema— that is, a higher-level representation of the data observations — you can type-check the observation to ensure that it matches the schema and the user’s expectation for later data analysis. For example, you can ensure that a number is contained at a given field location instead of a string given that you intend to perform numerical operations on it. A schema can also tell you whether the proper number of fields is represented for each observation (otherwise, your understanding of the data set may be flawed).

Some data-cleansing applications permit the construction of rules with functions that permit more complex transformations of data — for example, interrogating fields to create or modify other fields based on their contents. The rules can also validate the consistency of an observation (where possible) to remove invalid data or to transform data for greater accuracy — for example, modifying a U.S. ZIP code from five digits to the enhanced nine digits.

You can also identify duplicates, although there are applications for duplicate observations in a data set so that duplicate elimination isn't always required or necessary.

When a data set is syntactically correct, you can apply methods to ensure that the data is semantically correct. We'll explore one approach to this step next.

Data profiling

When your data is clean, the next step is to profile the data as a secondary step in the cleansing process. Profiling is an analysis of the data to ensure that the data is consistent. Through profiling,
you can dig into the data to see the distribution of the individual fields to look for outliers and other data that doesn't match the general data set.

For example, Figure 5 illustrates this process. In line 1, given that the real values represent physical measurements, a zero value may indicate an issue with this observation. In line 3, you see that the range of the measurement is obviously not in the same range as other measurements of this field (and its type differs). Finally, in line 5, notice that the class name is misspelled.

**Figure 5. Data set errors made visible through data profiling**

In some cases, these issues can be detected automatically through profiling. You could indicate that all measurements should be greater than 0 to catch the first issue. Through statistical analysis, you could identify the second outlier measurement. The final issue could be identified by capturing the unique class names and through their frequencies understand that this particular class name is an outlier (likely an occurrence of one).

You can validate time-series data in the context of flow to ensure that the data is processed in the correct order given timestamps. Otherwise, the validity of the result could be in question. You can use moving data windows to modify outliers (or *spikes*) to the window median value (as applied through Hampel filters).

Data profiling can be performed manually, which I'll demonstrate through the construction of a data-cleansing tool.

**Build a data-cleansing tool**

It’s relatively simple to build a generalized data-cleansing tool. I demonstrate the construction of a simple CSV data-cleansing tool that implements a set of key features, including:

- Field type checking through a user-defined schema
• Observation validation for too few or too many fields
• Automatic extraction of erroneous data for user review
• Training and test data generation using a user-defined probability
• Data summarization for simple data profiling

Let’s begin with a short discussion of the flow of this tool. Figure 6 illustrates the processes at work and the data that results.

**Figure 6. Flow for a data-cleansing and profiling tool**

I begin with a raw data set formatted as a comma-delimited text file. The user provides the schema as a string, identifying the type of each field used to parse and validate each field of each observation. Those that don’t pass the type check (or contain too few or too many fields) are written to an error file (which I can repair manually and reapply to the raw data set). Those observations that pass the initial checks are considered validated and move on to data set split and summarization. I can send each new sample to a training or test data file (based on a user-defined probability), and each field is summarized in a type-specific way. For example, strings are aggregated (up to 10), and integer and real values are profiled for minimum, maximum, and average.

You can find the source for the **simple cleansing tool** on GitHub, as shown below. This repository contains three source files that implement the input file management, line parsing, and summarization. The source splits the functionality between processing loop (`cleanse.c`), field parsing (`parse.c`), and summarization (`summary.c`). The processing loop parses the options from the command line, opens (and closes) all relevant files, and iterates over reading lines from the input file. These lines are passed to the parser, which then parses each individual field from the line and validates it by using the provided schema. Summaries are managed in a separate file to maintain a list of unique strings for a given field or profiling data for numerical data.
Figure 7. Flowchart illustrating the data-cleansing and profiling tool

Now let's look at the tool in action.

I'm using an old data set called the Zoo database created by Richard Forsyth that contains 15 binary-valued attributes describing an animal, with its name and class (called type) and a nonbinary attribute indicating the number of that animal's legs. The schema and sample are shown in Figure 8, with three of the 101 total observations (of seven possible classes) in this data set.
The source includes a `Makefile` I can use to build the tool simply by typing `make` on a standard Linux command line:

```bash
$ make
gcc -o cleanse cleanse.c parse.c summary.c -I. -g -std=c99 -Wall
```

The resulting executable is called `cleanse`. I invoke this tool by specifying the input file (the `-i` option), the output file (the `-o` option), and the schema (the `-c` option). The schema is represented by a single letter for each comma-delimited field in the data (`d` for integer, `s` for string, and `g` for float or double).

```bash
$ ./cleanse -i zoo.dat -o output -c "sdddddddddddddddd"
```

If I inspect the error file, I see the records that weren't correctly parsed and were therefore stored as errors. Note that comments are also generated to indicate where the first error was located in a given observation. As shown in the following code, the first error finds a question mark (?) in place of a binary value, and the second identifies a missing value. The final error lacks any data. For any data that had obvious issues, I could repair it in the original data set.

```bash
$ cat output.err
# Error in field 4
cat,1,0,0,?,0,0,1,1,1,1,0,0,4,1,1,1,1
# Error in field 4
dog,1,0,,0,0,1,1,1,1,0,0,4,1,1,1,1
# Error in field 0
,,,,,,,,,,,,,,,,,,
```

The output file (called `output.dat`) contains the observations the tool has validated:

```bash
$ more output.dat
aardvark,1,0,0,1,0,0,1,1,1,0,0,4,0,0,1,1
antelope,1,0,0,1,0,0,0,1,1,1,0,0,4,1,0,1,1
bass,0,0,1,0,0,1,1,1,1,0,0,1,0,1,0,0,4
...  
worm,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,7
wren,0,1,1,0,1,0,0,0,1,1,0,0,2,1,0,0,2
```

Cleansing, processing, and visualizing a data set, Part 1: Working with messy data
Using the valid data set, I can then view the profile of the data by specifying the `-p` option. A portion of this output is shown below, with the summary for the string data (the ellipsis `...` indicates that there were 10 or more unique strings) and three of the numeric fields. I can see from this data, for example, that there are few observations with feathers.

```
$ ./cleanse -i output.dat -o output2 -c "sddddddddddddd" -p

Data Profile:

Field 0: (STRING)
    aardvark
    antelope
    bass
    bear
    boar
    buffalo
    calf
    carp
    catfish
    cavy
    ...

Field 1: (INT)
    Min: 0
    Max: 1
    Avg: 0.425743

Field 2: (INT)
    Min: 0
    Max: 1
    Avg: 0.19802
    ...

Field 17: (INT)
    Min: 1
    Max: 7
    Avg: 2.83168
```

The final feature in this simple data-cleansing tool is the ability to split the data set into two parts: one for training and one for test. I can do this with the `-s` option (for **split** data set). To illustrate this function, I specify a 90/10 split of the data (using a probability of 0.9 for the training data), and then look at the line lengths for each output file (using the `wc`, or **word count** utility, and the `-l`, or **line** option) to see that the training and test data has been split roughly 90/10. Finally, I emit the 12-line test data file to see the distribution of the test observations.
I can now apply the training data file (output.dat) and the test data file (output.tst) to a machine learning algorithm, which I'll explore in the next tutorial for two unique approaches to classification.

Open source data-cleansing tools

You can find data-cleansing tools in the open source domain. One interesting example is called **Drake**, which performs data cleansing for text-based data by using a workflow approach that automatically handles dependencies in the available data and the commands to cleanse them. It supports multiple input and output files and has a similar operation as the `make` utility (in the context of managing dependencies).

The **DataCleaner tool** is a framework and data-profiling engine that exposes an API and allows user-defined extensions for data cleansing. DataCleaner supports multiple input and output formats, with the ability to create rules for data quality over your data.

Going further

This tutorial explored some problems and solutions for cleansing data, and discussed the development of a simple utility that can perform data validation and profiling. In Part 2, I'll take this cleansed data and build two models (based on adaptive resonance theory and vector quantization) to classify the data and quantify the accuracy of each model. In Part 3, I'll explore some of the ways you can visualize this data using a variety of open source tools.
Related topics

- Register for a Cloud Lite account
- Subscribe to the Data and Analytics newsletter
- Try a code pattern

© Copyright IBM Corporation 2017

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)