The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets

Nick Hynes*
Berkeley AI Research (BAIR) Lab
nhynes@berkeley.edu

D. Sculley
Google Brain
dsculley@google.com

Michael Terry
Google Brain
michaelterry@google.com

Abstract

Data cleaning and feature engineering are both common practices when developing machine learning (ML) models. However, developers are not always aware of best practices for preparing or transforming data for a given model type, which can lead to suboptimal representations of input features. To address this issue, we introduce the *data linter*, a new class of ML tool that automatically inspects ML data sets to 1) identify potential issues in the data and 2) suggest potentially useful feature transforms, for a given model type. As with traditional code linting, data linting automatically identifies potential issues or inefficiencies; codifies best practices and educates end-users about these practices through tool use; and can lead to quality improvements. In this paper, we provide a detailed description of data linting, describe our initial implementation of a data linter for deep neural networks, and report results suggesting the utility of using a data linter during ML model design.

1 Introduction: Lints and Data Lints

The concept of a *lint* originates in the field of software engineering, where it refers to a potential defect in the expression of a program, or a deviation from accepted practices [10]. Canonical examples of code lint include unused code, the use of bug-prone constructs or idioms, and stylistic patterns that do not conform to a community-accepted standard. Importantly, a linter warning does not necessarily indicate an error exists. However, it *does* reduce the cognitive burden of consistently applying best practices, and of identifying potential bugs in the code.

As machine learning (ML) systems have become widely deployed, it has become clear that ML data sets are an area in which errors and bugs can have far reaching consequences on system behavior [16]. Indeed, ML data takes on the role of ML code in many respects, which makes testing, verification, and data cleaning critical tasks.

A large number of tests for ML data have been developed [4] and are critical for production systems. We propose that many *data* issues can be identified with lightweight automation—at least as potential issues that warrant further investigation. Some examples include:

- Numerical features on widely differing scales.
- Specific values ascribed special meaning (e.g., -9999 to represent missing data).
- Malformed values of special string types, such as dates.

This list is naturally non-exhaustive, but these problems are all too familiar to practitioners working with real data, especially when a human is the source of the data.

Catching such problems in ML data sets is commonly performed as part of cleaning which, even when automated, is a time-consuming and error-prone process of repeated inspection and correction [5].

*Work was performed while at Google.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

To alleviate this issue, we propose a general-purpose tool for this task: a *data linter*. The data linter is a tool that analyzes a user's training data and suggests ways features can be transformed to improve model quality, for a specific model type. To identify potential issues, the data linter inspects the training data's summary statistics, examines individual examples, and considers the names developers give to their features. For each potential issue identified, the data linter produces a warning, a recommendation for how to improve the feature's representation, and a concrete example of the lint taken directly from the data.

Because this is a new class of tool in the generic ML toolkit, this paper makes a number of contributions. We introduce and define the concept of data linting, and describe a taxonomy of data lint that focuses on deep neural networks (DNNs) as a target model type. We present the high-level architecture and implementation of our data linter tool, noting that the design allows third parties to add new data lint detectors. Finally, we provide an empirical evaluation, reporting results of applying this tool to a set of 600 publicly available data sets from Kaggle, in addition to several proprietary data sets internal to Google. Our results suggest that the data linter can be valuable to developers who are new to machine learning, as both an educational tool and as a means to improve model quality.

2 Related Work: Code Linters, Data Cleaning, and Automated ML

The concept of a data linter builds upon work in code linters; data cleaning and data validation; and automated model construction and automated feature engineering.

Code linters (e.g., Pylint [14], ESLint [7]) inspect code and output warnings and recommendations for software developers. These recommendations are intended to help make the code more readable by humans, and to prevent the introduction of bugs through the use of problematic idioms. Importantly, "linty" code and "non-linty" code can be functionally equivalent to the interpreter or compiler. In these cases, the only difference is that lint-free code conforms to a set of established best practices, which are intended to make the code more human-interpretable and/or consistent.

A data linter is very much in the spirit of code linters. Some feature representations are very natural to people, but suboptimal to specific ML models. For instance, a person might prefer to use the values 0-360 to represent an angle in degrees, but a machine learning model might learn better using a bucketized or sine-transformed value. Crucially, the data is valid, in that the model will accept the inputs, but it is not in a representation the model can best learn from. The data linter identifies such potential issues.

Data cleaning and validation have long histories in the domains of databases, data warehousing, data science, and machine learning, among many others. Data cleaning includes a diverse set of activities, including normalizing the data (e.g., correcting misspellings), removing data with illegal values (e.g., NaN values), identifying outliers, and automatically correcting problematic instances (see [5, 6, 15] for surveys of the space). Data validation ensures that data conforms to a schema or a set of constraints, and is becoming a standard tool in ML pipelines [2, 13]. For example, data validation can ensure that numerical values fall within a specific range, or that string values are limited to a prescribed set of values. These processes can be fully automated [2, 12], or include a human-in-the-loop [3, 11] for increased flexibility.

In the context of ML, data validation is essential to ensure that an ML model will function correctly [2]. Additionally, data cleaning can increase the quality of the trained model (e.g., by fixing incorrect features/labels). However, data can be both cleaned and validated, but still have a suboptimal representation for a given class of model. For example, although a linear classifier can accept numeric features in any range, it may yield higher quality results if all of the inputs are normalized with respect to each other.

Automated model construction, including automated feature engineering, has the goal of minimizing the amount of effort required to produce a high quality model (e.g., [8, 17]). This is an active research area producing promising results, but it is still an evolving area. For those who must manually define models, the data linter can provide useful advice and assistance for feature engineering.

3 Data Linter Design and Implementation

As mentioned in the Introduction, we define a *data linter* as a tool that analyzes training data features, with respect to a specific model type, and provides recommendations for how individual features

can be transformed to increase the likelihood that the model can learn from them. In this context, a *data lint* is a data- and model-specific inspection rule and recommendation. For example, a "date-time" data lint examines data to see whether it is likely a date or time encoded as a string. If so, it recommends conversion to a numeric timestamp.

Our implementation of a data linter includes the following components:

- A user-extensible collection of *LintDetectors*. A LintDetector corresponds to a specific issue to search for, for a given model type. LintDetectors thus include the logic to detect the problem, as well as code to collect sample instances to present to the user later, to help them understand and debug identified issues. In the next section, we describe the different types of information we found useful to implement LintDetectors.
- A *DataLinter*, or the engine that applies the LintDetectors to a data set.
- A *LintExplorer*, which takes the output of the DataLinter and presents it to the user. For every lint detected, a detailed explanation of the issue (including suggested fixes) is presented, along with sample data illustrating the issue in the user's data set.
- Data summarization functions to calculate summary statistics over the data set, for LintDetectors that need summary statistics to operate.

Creating a new LintDetector amounts to subclassing the LintDetector class and implementing a lint method that accepts a collection of data instances and returns a *LintResult* that records what features triggered warnings, along with sample instances.

3.1 DataLinter Implementation for DNNs

Our implementation of the DataLinter for DNNs leverages the Apache Beam framework [1] to load the dataset and run each LintDetector. Each LintDetector runs independently to afford a degree of scalability.

Since we envision using the DataLinter as part of a larger ML pipeline, it exists as a Python library with an optional standalone frontend. When integrated into a larger system, data linting most logically occurs after both the data ingestion and data pre-processing stages.

To identify potential problems, we found it useful to make the following types of information available to individual LintDetectors: 1) summary statistics calculated over the entire data set (e.g., to identify outliers), 2) individual data instances (e.g., to identify long strings), and 3) feature names and metadata.

To reduce computational burden, we use pre-computed statistics and, where possible, heuristics based on them. As would be expected, LintDetectors that scan all examples (e.g., to detect duplicates or enumerations) run several orders of magnitude more slowly than those that simply observe histogram summaries of data (i.e. tailed distribution, uncommon sign/list length). As we note in section 5, the prototype implementation can be slowed down significantly by the presence (but, notably, not quantity) of such LintDetectors. Fortunately, improving performance is an issue of implementation, not design. For instance, duplicate/enum detection can be made to run in logarithmic rather than linear time and space simply by replacing hashing with the HyperLogLog algorithm [9].

The feature name can be useful to identify data types such as zip codes or geospatial data. While it can be difficult to discern whether numeric data is latitudinal or longitudinal through inspecting numerical values alone, it is quite likely that the feature name includes a "lat" or "lon" string. The same is true of zip codes, as we found in our evaluation of Kaggle data sets.

Our initial prototype of the DataLinter is available under a free and open source license [2].

4 Data Lints

In this section, we detail a number of specific lints derived from common errors observed within Google and through inspection of publicly available data sets. These lints can be roughly classified into three high-level categories: miscodings of data, outliers, and packaging errors. Note that while we consider data lints to be tied to specific model types (since feature engineering recommendations

[2] `https://github.com/brain-research/data-linter`

are generally model-dependent), we acknowledge that some data lints are more broadly applicable (e.g., unnormalized features). This observation suggests the potential for reuse of data lint detectors across model types.

4.1 Miscoding Lints

Miscoding data linters attempt to identify data that should be transformed to improve the likelihood that a model can learn from the data. While some model types like Random Forests may be robust to some of these lints, many of the lints described below are applicable to a range of model types, including neural networks, linear models, and SVMs.

In our work, we developed the following miscoding data lint rules:

- Number as string: A number is encoded as a string. Consider whether it should be represented as a number.
- Enum as real: An enum (a categorical value) is encoded as a real number. Consider converting to an integer and using an embedding or one-hot vector.
- Tokenizable string: A feature has very long strings that are likely all unique. Consider tokenizing and using word embeddings.
- Circular domain as linear: A circular feature was identified (e.g., latitude, longitude, day of the week). Consider bucketing the value or transforming it via a trigonometric function.
- Date/time as a string: Consider encoding as a timestamp.
- Zip code as number: A zip code should probably be bucketed or represented as an embedding.
- Integer as float: Integers are encoded as floats. Consider whether they may actually be enums.

4.2 Lints for Outliers and Scaling

Outlier and scaling data linters attempt to identify likely outliers and scaling issues in data.

- Unnormalized feature: The values of this feature vary widely. Consider normalizing them.
- Tailed distribution detector: Extreme values that significantly affect the mean were detected. Examine histograms of the data to ensure they follow expected distributions.
- Uncommon list length: A feature is composed of a list of elements, with most instances consisting of a specific list length. However, some instances have a different length.Ensure that the data are materialized as expected and the model can handle data lists of varying length.
- Uncommon sign detector: The data includes some values that have a different sign (+/-) from the rest of the data (e.g., -9999), which can affect training. If these are special markers in the data, consider replacing them with a more neutral value (e.g., an empty or average value).

4.3 Packaging Error Lints

Packaging error data linters identify problems with the organization of the data. Some of these issues may be caught in an early data cleaning or validation step.

- Duplicate values: Duplicate rows of data have been identified. Verify that there is no error in data generation.
- Empty examples: Empty examples have been detected. Consider removing them and verifying correctness of your data generation process. (Note that the *entire* example must be empty; empty *features* suggest correct use of missing-value semantics.)

5 Evaluation

To evaluate our data linter, we performed two studies: We invited eight software engineers to try the data linter on the training data of a model they were developing, and we ran the data linter on 600 publicly available Kaggle data sets.

5.1 End-User Evaluation

Eight different software engineers applied the data linter to training data for models under development. These software engineers ranged from interns who were relatively new to machine learning (and were tasked with developing new models), to a software engineer working with mature machine learning models. In all cases, the participants were instructed on use of the tool, then interviewed to understand the overall utility of the tool and its results.

In the most promising result, a data linter suggestion (normalize inputs) led to a DNN model's precision increasing from 0.48 to 0.59; this improvement was after the engineer had already automatically tuned the model's hyperparameters. This user was unaware of the benefits of normalizing inputs to a DNN, so the tool also served as an educational aid. The data linter helped another developer identify duplicate training data that had previously gone unnoticed.

In many cases, the linter's suggestions prompted further analysis and consideration of the feature engineering choices made for their data, vis-a-vis their chosen model, suggesting its utility in focusing development and debugging efforts.

One common request from users was to be able to silence lints, either for specific inputs or specific lint types. This is a common feature in other linters, and would be useful for this tool, as well. Furthermore, in some cases, the user may already have proper feature transformations in their model, meaning the linter warnings are unnecessary. Accordingly, data linting may be useful both before and after feature transformation, with lints "fixed" by feature transforms being removed prior to presentation to the user.

In terms of performance, for medium-sized datasets of $O(100k)$ examples, we observed that constant time/space (i.e. stats-based) LintDetectors completed in seconds; linear time, sub-linear space Detectors finished in minutes; and linear space/time Detectors (i.e. duplicate/enum) took up to several hours (these timings are with a few CPUs running in parallel). With the exception of the latter LintDetectors, which use a particularly inefficient hashing technique, we find that straightforward implementations perform reasonably well, at scale. We also note that the set of LintDetectors that run can be configured by the user in accordance with their tolerance for latency.

5.2 Kaggle Data Set Analysis

To test the linter on a broader set of data, we worked with Kaggle to obtain 600 publicly available CSV data sets. We loaded each CSV into a table using the popular Python `pandas` package, and used its default, automatic data type inference capability (mimicking the workflow of novice ML users). This use case also provided an ideal context for testing a data linter: `pandas`' column type inference algorithm always chooses the most flexible type (e.g., a misplaced string in an otherwise int column yields a string column), meaning the data linter should automatically uncover data in formats that are less than ideal for a model. During development of the lints, we used 40 of these data sets to help inform the types of data lints to create.

The most common lints triggered, and the number of lints per data set, can be viewed in figs. 1 and 2.

Overall, only about 7% of the data sets in this study had no data lints, and most data sets presented multiple data lints. These findings may include some false positives resulting from the fact that data types were inferred by `pandas`, rather than explicitly declared. In aggregate, however, these results suggest the utility of automated data linting for focusing developer effort during preparation of data for ML models. Additionally, we note that the number of lints identified is still manageable, with a mode of about 4 lints per data set; users are unlikely to be overwhelmed with this amount of information. For datasets with thousands of features and repetitive detections, it could be worthwhile to modify the lint output so users can view summaries of detected lint classes.

To determine the prevalence of issues that were not identified by our tool, we randomly sampled and manually examined 35 data sets (these were not previously referenced when defining lints). We found no false negatives, but acknowledge that the validity of this analysis is limited by our own knowledge of what *could* be considered problematic for a generic ML system.

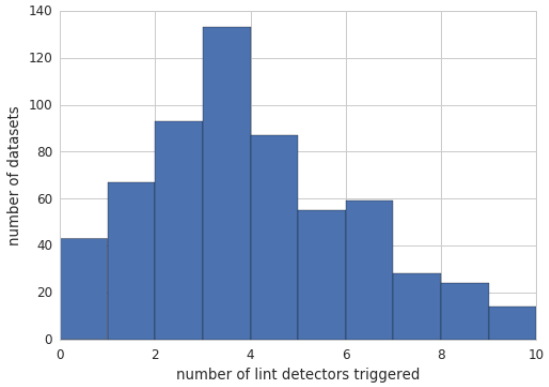

Figure 1: **Histogram of Number of Data Lints Per Data Set, Across 600 Kaggle Data Sets.** Interestingly, only about 7% of the data sets we examined as part of this study had zero data lints and were thus "lint free." The other 93% had at least one data lint, suggesting the utility of automated checking.

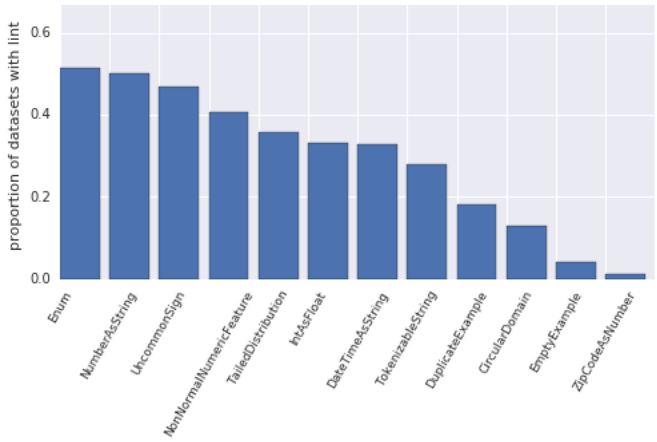

Figure 2: **Frequency of Data Lints Across 600 Kaggle Data Sets.** Some data issues are relatively common, including encoding numeric values as strings and issues around encoding enumerated values. But even rarer issues such as empty examples or potentially problematic encoding of postal codes do occur in the real-world data sets.

6 Conclusion and Future Work

Our development of a data linter suggests that the concept and tool can be useful to identify ways model quality could be improved through specific feature transforms, and to help focus developer effort and attention.

There are a number of directions in which this work may be extended. These include implementing more scalable algorithms in current lint detectors and developing new detectors for both DNNs and other model types. Furthermore, the ability to automatically infer the semantic meaning/intent of a feature could also plug directly into a tool for (semi-)automated model construction (e.g., [8, 17]). In this case, the data linter could inspect the training data to produce specific feature engineering recommendations that can be automatically applied during model construction.

Acknowledgments

We thank Jimbo Wilson for conversations that led to this research. We also thank the participants who tested the data linter.

References

[1] *Apache Beam*. https://beam.apache.org. Accessed: 2017-10-23.

[2] Denis Baylor et al. "The Anatomy of a Production-Scale Continuous-Training Machine Learning Platform". In: *ACM Conference on Knowledge Discovery and Data Mining*. 2017.

[3] Moria Bergman et al. "Query-Oriented Data Cleaning with Oracles". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1199–1214. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2737786. URL: http://doi.acm.org/10.1145/2723372.2737786.

[4] Eric Breck et al. "What's your ML test score? A rubric for ML production systems". In: *Reliable Machine Learning in the Wild (NIPS 2016 Workshop)*. 2016.

[5] Xu Chu et al. "Data Cleaning: Overview and Emerging Challenges". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 2201–2206. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2912574. URL: http://doi.acm.org/10.1145/2882903.2912574.

[6] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2003. ISBN: 0471268518.

[7] *ESLint*. http://eslint.org/about. Accessed: 2017-10-20.

[8] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf.

[9] Philippe Flajolet et al. "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm". In: *AofA: Analysis of Algorithms*. Discrete Mathematics and Theoretical Computer Science. 2007, pp. 137–156.

[10] S. C. Johnson. "Lint, a C Program Checker". In: *COMP. SCI. TECH. REP*. 1978, pp. 78–1273.

[11] Sean Kandel et al. "Wrangler: Interactive Visual Specification of Data Transformation Scripts". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: ACM, 2011, pp. 3363–3372. ISBN: 978-1-4503-0228-9. DOI: 10.1145/1978942.1979444. URL: http://doi.acm.org/10.1145/1978942.1979444.

[12] Sanjay Krishnan et al. "ActiveClean: Interactive Data Cleaning for Statistical Modeling". In: *Proc. VLDB Endow.* 9.12 (Aug. 2016), pp. 948–959. ISSN: 2150-8097. DOI: 10.14778/2994509.2994514. URL: http://dx.doi.org/10.14778/2994509.2994514.

[13] Alkis Polyzotis et al. "Data Management Challenges in Production Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. New York, NY, USA, 2017, pp. 1723–1726.

[14] *Pylint*. http://www.pylint.org. Accessed: 2017-10-20.

[15] Erhard Rahm and Hong Hai Do. "Data Cleaning: Problems and Current Approaches". In: *IEEE Data Engineering Bulletin* 23 (2000), p. 2000.

[16] D. Sculley et al. "Machine Learning: The High Interest Credit Card of Technical Debt". In: *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*. 2014.

[17] Chris Thornton et al. "Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms". In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: ACM, 2013, pp. 847–855. ISBN: 978-1-4503-2174-7. DOI: 10.1145/2487575.2487629. URL: http://doi.acm.org/10.1145/2487575.2487629.