Datalab: A unified audit to detect all kinds of issues in data and labels

↑ Back to top

Run in Google Colab

Cleanlab offers a `Datalab` object that can identify various issues in your machine learning datasets, such as noisy labels, outliers, (near) duplicates, drift, and other types of problems common in real-world data. These data issues may negatively impact models if not addressed. `Datalab` utilizes *any* ML model you have already trained for your data to diagnose these issues, it only requires access to either: (probabilistic) predictions from your model or its learned representations of the data.

Overview of what we'll do in this tutorial:

- Compute out-of-sample predicted probabilities for a sample dataset using cross-validation.
- Use `Datalab` to identify issues such as noisy labels, outliers, (near) duplicates, and other types of problems
- View the issue summaries and other information about our sample dataset

You can easily replace our demo dataset with your own image/text/tabular/audio/etc dataset, and then run the same code to discover what sort of issues lurk within it!

Quickstart

Already have (out-of-sample) `pred_probs` from a model trained on an existing set of labels? Maybe you also have some numeric `features` (or model embeddings of data)? Run the code below to examine your dataset for multiple types of issues.

```
from cleanlab import Datalab

lab = Datalab(data=your_dataset, label_name="column_name_of_labels")
lab.find_issues(features=your_feature_matrix, pred_probs=your_pred_probs)

lab.report()
```

1. Install and import required dependencies

`Datalab` has additional dependencies that are not included in the standard installation of cleanlab.

You can use pip to install all packages required for this tutorial as follows:

```
!pip install matplotlib
!pip install "cleanlab[datalab]"

# Make sure to install the version corresponding to this tutorial
# E.g. if viewing master branch documentation:
#     !pip install git+https://github.com/cleanlab/cleanlab.git
```

↑ Back to top

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict

from cleanlab import Datalab
```

2. Create and load the data (can skip these details)

We'll load a toy classification dataset for this tutorial. The dataset has two numerical features and a label column with three possible classes. Each example is classified as either: *low*, *mid* or *high*.

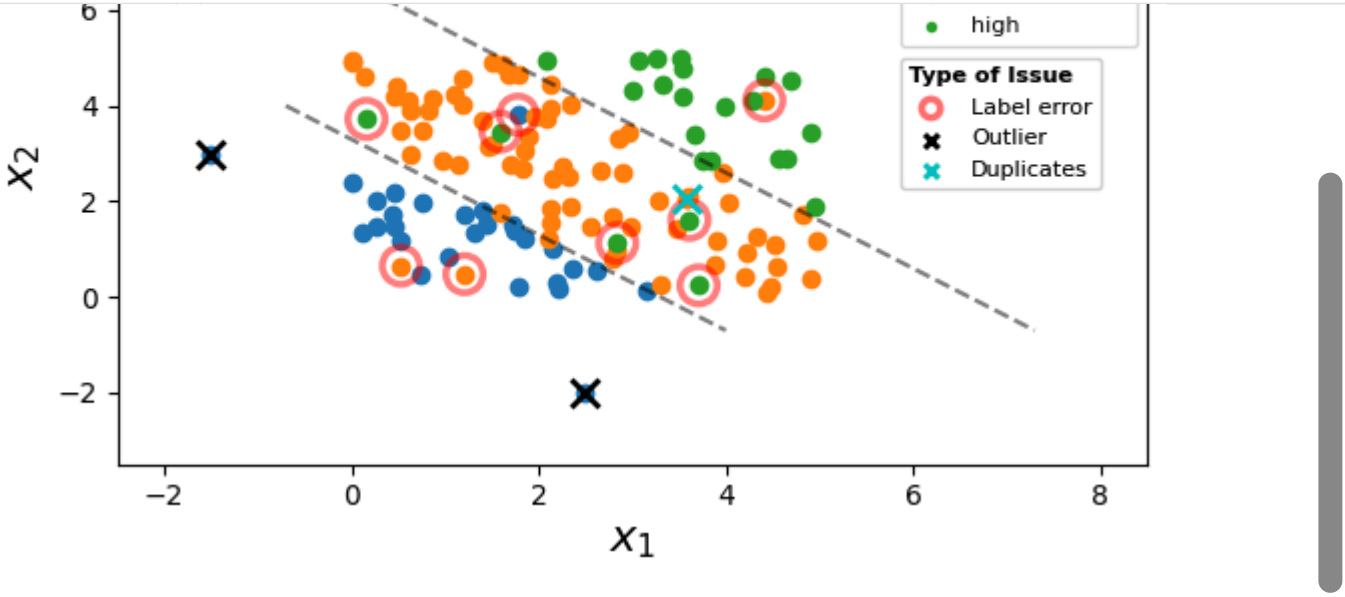
▶ See the code for data generation. **(click to expand)**

```
X_train, y_train_idx, noisy_labels, noisy_labels_idx, X_out, X_duplicate = create_data(
```

We make a scatter plot of the features, with a color corresponding to the observed labels. Incorrect given labels are highlighted in red if they do not match the true label, outliers highlighted with an a black cross, and duplicates highlighted with a cyan cross.

▶ See the code to visualize the data. **(click to expand)**

```
plot_data(X_train, y_train_idx, noisy_labels_idx, X_out, X_duplicate)
```



In real-world scenarios, you won't know the true labels or the distribution of the features, so we won't use these in this tutorial, except for evaluation purposes.

`Datalab` has several ways of loading the ase, we'll simply wrap the training features and noisy labels in a dictionary so that we can pass it to `Datalab`.

↑ Back to top

```
data = {"X": X_train, "y": noisy_labels}
```

Other supported data formats for `Datalab` include: HuggingFace Datasets and pandas DataFrame. `Datalab` works across most data modalities (image, text, tabular, audio, etc). It is intended to find issues that commonly occur in datasets for which you have trained a supervised ML model, regardless of the type of data.

Currently, pandas DataFrames that contain categorical columns might cause some issues when instantiating the `Datalab` object, so it is recommended to ensure that your DataFrame does not contain any categorical columns, or use other data formats (eg. python dictionary, HuggingFace Datasets) to pass in your data.

3. Get out-of-sample predicted probabilities from a classifier

To detect certain types of issues in classification data (e.g. label errors), `Datalab` relies on predicted class probabilities from a trained model. Ideally, the prediction for each example should be out-of-sample (to avoid overfitting), coming from a copy of the model that was not trained on this example.

This tutorial uses a simple logistic regression model and the `cross_val_predict()` function from scikit-learn to generate out-of-sample predicted class probabilities for every example in the training set. You can replace this with *any* other classifier model and train it with cross-validation to get out-of-sample predictions. Make sure that the columns of your `pred_probs` are properly ordered with respect to the ordering of classes, which for Datalab is: lexicographically sorted by class name.

```
model = LogisticRegression()
pred_probs = cross_val_predict(
    estimator=model, X=data["X"], y=data["y"], cv=5, method="predict_proba"
)
```

4. Use Datalab to find issues in the dataset

We create a `Datalab` object from the dataset, also providing the name of the label column in the dataset. Only instantiate one `Datalab` object per dataset, and note that only classification datasets are supported for now.

All that is need to audit your data is to call `find_issues()`. This method accepts various inputs like: predicted class probabilities, numeric feature representations of the data. The more

information you provide here, the more thoroughly `Datalab` will audit your data! Note that `features` should be some numeric representation of each example, either obtained through preprocessing transformation of your raw data or embeddings from a (pre)trained model. In this case, our data is already entirely numeric so we just provide the features directly.

↑ Back to top

```
lab = Datalab(data, label_name="y")
lab.find_issues(pred_probs=pred_probs, features=data["X"])
```

```
Finding null issues ...
Finding label issues ...
Finding outlier issues ...
Finding near_duplicate issues ...
Finding non_iid issues ...
Finding class_imbalance issues ...
Finding underperforming_group issues ...

Audit complete. 30 issues found in the dataset.
```

Now let's review the results of this audit using `report()`. This provides a high-level summary of each type of issue found in the dataset.

```
lab.report()
```

```
Dataset Information: num_examples: 132, num_classes: 4

Here is a summary of various issues found in your data:

     issue_type  num_issues
          label          17
        outlier           6
 near_duplicate           4
class_imbalance           3

Learn about each issue: https://docs.cleanlab.ai/stable/cleanlab/datalab/guide/issue_
See which examples in your dataset exhibit each issue via: `datalab.get_issues(<ISSUI

Data indices corresponding to top examples of each issue are shown below.


---------------------- label issues ----------------------

About this issue:
        Examples whose given label is estimated to be potentially incorrect
```

5. Learn more about the issues in your dataset

Datalab detects all sorts of issues in a dataset and what to do with the findings will vary case-by-case. For automated improvement of a dataset via best practices to handle auto-detected issues, try Cleanlab Studio.

To conceptually understand how each type of issue is defined and what it means if detected in your data, check out the Issue Type Descriptions page. The Datalab Issue Types page also lists additional types of issues that `Datalab.find_issues()` can detect, as well as optional parameters you can specify for greater control over how your data are checked.

Datalab offers several methods to understand more details about a particular issue in your dataset. The `get_issue_summary()` method fetches summary statistics regarding how severe

each type of issue is overall across the whole dataset.

```
lab.get_issue_summary()
```

↑ Back to top

	issue_type	score	num_issues
0	null	1.000000	0
1	label	0.856061	17
2	outlier	0.355772	6
3	near_duplicate	0.616034	4
4	non_iid	0.821750	0
5	class_imbalance	0.022727	3
6	underperforming_group	0.926818	0

In the returned summary DataFrame: LOWER `score` values indicate types of issues that are MORE severe *overall* across the dataset (lower-quality data in terms of this issue), HIGHER `num_issues` values indicate types of issues that are MORE severe *overall* across the dataset (more datapoints appear to exhibit this issue).

We can also only request the summary for a particular type of issue.

```
lab.get_issue_summary("label")
```

	issue_type	score	num_issues
0	label	0.856061	17

The `get_issues()` method returns information for each *individual example* in the dataset including: whether or not it is plagued by this issue (Boolean), as well as a *quality score* (numeric value betweeen 0 to 1) quantifying how severe this issue appears to be for this particular example.

```
lab.get_issues().head()
```

	is_null_issue	null_score	is_label_issue	label_score	is_outlier_issue	outlier_score	is_near_duplicate_
0	False	1.0	False	0.859131	False	0.417707	
1	False	1.0	False	0.816953	False	0.375317	
2	False	1.0	False	0.531021	False	0.460593	
3	False	1.0	False	0.752808	False	0.321635	
4	False	1.0	True	0.090243	False	0.472909	

Each example receives a separate *quality score* for each issue type (eg. `outlier_score` is the *quality score* for the `outlier` issue type, quantifying *how typical* each datapoint appears to be). LOWER scores indicate MORE severe instances of the issue, so the most-concerning datapoints have the lowest quality scores. Sort by these scores to see the most-concerning

examples in your dataset for each type of issue. The quality scores are directly comparable between examples/datasets, but not across different issue types.

↑ Back to top

Similar to above, we can pass the type c ument to `get_issues()` to get the information for one particular type of issue. As an example, let's see the examples identified as having the most severe *label* issues:

```
examples_w_issue = (
    lab.get_issues("label")
    .query("is_label_issue")
    .sort_values("label_score")
)

examples_w_issue.head()
```

	is_label_issue	label_score	given_label	predicted_label
7	True	0.008963	low	mid
120	True	0.009664	high	mid
40	True	0.013445	mid	low
107	True	0.025184	high	mid
53	True	0.026376	high	mid

Inspecting the labels for some of these top-ranked examples, we find their given label was indeed incorrect.

Get additional information

Miscellaneous additional information (statistics, intermediate results, etc) related to a particular issue type can be accessed via `get_info(issue_name)`.

```
label_issues_info = lab.get_info("label")
label_issues_info["classes_by_label_quality"]
```

	Class Name	Class Index	Label Issues	Inverse Label Issues	Label Noise	Inverse Label Noise	Label Quality Score
0	low	1	12	2	0.428571	0.111111	0.571429
1	high	0	11	2	0.407407	0.111111	0.592593
2	mid	3	25	5	0.337838	0.092593	0.662162
3	max	2	1	40	0.333333	0.952381	0.666667

This portion of the info shows overall label quality summaries of all examples annotated as a particular class (e.g. the `Label Issues` column is the estimated number of examples labeled as this class that should actually have a different label). To learn more about this, see the documentation for the cleanlab.dataset.rank_classes_by_label_quality method.

You can view all sorts of information regarding your dataset using the `get_info()` method with no arguments passed. This is not printed here as it returns a huge dictionary but feel free to check it out yourself! Don't worry if you don't understand all of the miscellaneous information

in this `info` dictionary, none of it is critical to diagnose the issues in your dataset. Understanding miscellaneous info may require reading the documentation of the miscellaneous cleanlab functions which

↑ Back to top

Near duplicate issues

Let's also inspect the examples flagged as (near) duplicates. For each such example, the `near_duplicate_sets` column below indicates *which* other examples in the dataset are highly similar to it (this value is empty for examples not flagged as nearly duplicated). The `near_duplicate_score` quantifies *how similar* each example is to its nearest neighbor in the dataset.

```
lab.get_issues("near_duplicate").query("is_near_duplicate_issue").sort_values("near_dup
```

	is_near_duplicate_issue	near_duplicate_score	near_duplicate_sets	distance_to_nearest_neighbor
123	True	0.000000	[131]	0.000000e+00
131	True	0.000000	[123]	0.000000e+00
129	True	0.000002	[130]	4.463180e-07
130	True	0.000002	[129]	4.463180e-07

Learn more about handling near duplicates detected in a dataset from the FAQ.

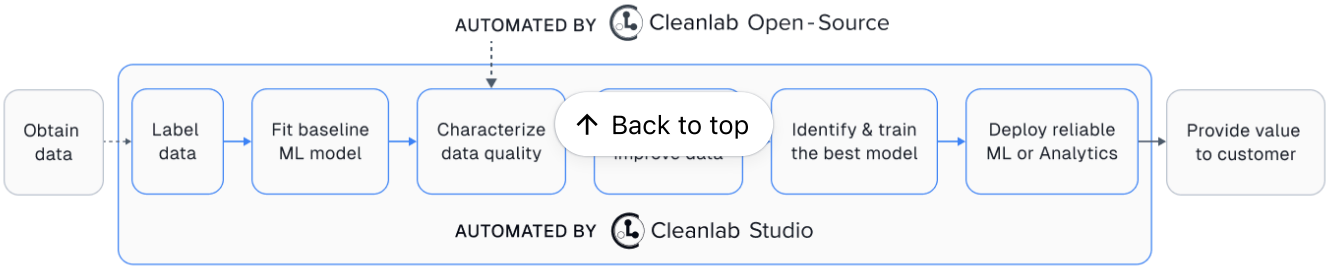
Other issues detected in this tutorial dataset include **outliers** and **class imbalance**, see the Issue Type Descriptions for more information. `Datalab` makes it very easy to check your datasets for all sorts of issues that are important to deal with for training robust models. The inputs it uses to detect issues can come from *any* model you have trained (the better your model, the more accurate the issue detection will be).

To learn more, check out this example notebook (demonstrates Datalab applied to a real dataset) and the advanced Datalab tutorial (demonstrates configuration and customization options to exert greater control).

Spending too much time on data quality?

Using this open-source package effectively can require significant ML expertise and experimentation, plus handling detected data issues can be cumbersome.

That's why we built Cleanlab Studio – an automated platform to find **and fix** issues in your dataset, 100x faster and more accurately. Cleanlab Studio automatically runs optimized data quality algorithms from this package on top of cutting-edge AutoML & Foundation models fit to your data, and helps you fix detected issues via a smart data correction interface. Try it for free!



↑ Back to top

`cleanlab` is distributed on PyPI and conda.

Made with Sphinx and @pradyunsg's Furo