Data Leakage in Notebooks: Static Detection and Better Processes

Chenyang Yang
Carnegie Mellon University

Rachel A Brower-Sinning
Carnegie Mellon Software Engineering Institute

Grace A. Lewis
Carnegie Mellon Software Engineering Institute

Christian Kästner
Carnegie Mellon University

ABSTRACT

Data science pipelines to train and evaluate models with machine learning may contain bugs just like any other code. Leakage between training and test data can lead to overestimating the model's accuracy during offline evaluations, possibly leading to deployment of low-quality models in production. Such leakage can happen easily by mistake or by following poor practices, but may be tedious and challenging to detect manually. We develop a static analysis approach to detect common forms of data leakage in data science code. Our evaluation shows that our analysis accurately detects data leakage and that such leakage is pervasive among over 100,000 analyzed public notebooks. We discuss how our static analysis approach can help both practitioners and educators, and how leakage prevention can be designed into the development process.

ACM Reference Format:
Chenyang Yang, Rachel A Brower-Sinning, Grace A. Lewis, and Christian Kästner. 2022. Data Leakage in Notebooks: Static Detection and Better Processes. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556918

1 INTRODUCTION

Will a promising machine-learned model work when deployed in production? Typically this question is answered by comparing model predictions to expected outcomes on test data. However, the resulting accuracy estimates can be misleading, where the model performs well on test data, but much worse in production. A common cause is that the data used for testing is not representative enough of the production data, thus providing misleading estimates on the wrong data distribution. A different cause, and the focus of this paper, is that the test data was used in some form during model training (directly or indirectly, intentionally or accidentally) allowing the model to overfit on the test data, thus producing unrealistically optimistic accuracy estimates. Because data science pipelines are code, we can use software engineering techniques to analyze them—which we do in this paper.

In this paper, we design a static analysis approach to detect cases where model training makes use of test data in data science code, commonly called *data leakage* [4, 19]. Data leakage is often the result of using bad practices when writing machine learning code, ranging from obvious mistakes, such as including test data in the training data, to more subtle ones that leak test data distribution information through preprocessing prior to training. For example, in Fig. 1, we show data science code reporting confidently to find patterns in random data where the model should not do better than a random guess: Because decisions during training depend on both training and test data (feature selection, Line 9) the model overfits on test data and the evaluation reports significantly inflated accuracy scores. Our analysis points out common pitfalls in model accuracy evaluations like the one in our example, which, as we will show, are pervasive in data science code in public notebooks.

```
1  import numpy as np
2  # generate random data
3  n_samples, n_features, n_classes = 200, 10000, 2
4  rng = np.random.RandomState(42)
5  X = rng.standard_normal((n_samples, n_features))
6  y = rng.choice(n_classes, n_samples)
7
8  # leak test data through feature selection
9  X_selected = SelectKBest(k=25).fit_transform(X, y)
10
11 X_train, X_test, y_train, y_test = train_test_split(
12     X_selected, y, random_state=42)
13 gbc = GradientBoostingClassifier(random_state=1)
14 gbc.fit(X_train, y_train)
15
16 y_pred = gbc.predict(X_test)
17 accuracy_score(y_test, y_pred)
18 # expected accuracy ~0.5; reported accuracy 0.76
```

Figure 1: Data leakage may cause a highly-biased test result. The model learns test data distribution through feature selection, resulting in an over-optimistic test score.

Our analysis has both practical and educational value. On the practical side, our work contributes to more reliable offline evaluations of machine-learned models, which are an important quality assurance step when integrating models into software products. Use of machine learning in software products is increasingly common, but also very challenging [2, 13, 29]. Reliable offline accuracy evaluations are important for preventing harm from deploying low-quality models in production systems, where harm can range from stress, to discrimination, to fatal accidents [31, 36]. Accuracy results are also a common quality metric between teams [29], especially when delegating or entirely outsourcing model development. When accuracy goals are parts of contracts (or competitions) there may be an incentive to report inflated accuracy results.

On the educational side, the danger of overfitting and data leakage is well known and commonly discussed in textbooks [4, 28], ML library documentation [15, 40], and tutorials [1]. Yet, as we will show, leakage also occurs in tutorial notebooks, popular notebooks,

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556918

and entries in data science competitions, which others may use as educational resources or templates. Our analysis, just like other static analyses, can help raise awareness of coding problems and nudge students and model developers toward better practices.

Technically, we develop a static data-flow analysis that tracks how datasets flow through data science code and are used in training and evaluation functions of machine-learning libraries. To allow accurate detection, we track specific kinds of transformations and detect common patterns that lead to leakage. In an evaluation with data science code from public notebooks, we show that our analysis is accurate (92.9%) with very few false positives and can analyze most notebooks within a few seconds. Applying our analysis to over 100,000 public notebooks, we detect data leakage issues in nearly 30 percent of them.

In summary, we make the following contributions:

- A summary and formulation of common data leakage problems.
- A static analysis that can automatically detect data leakage.
- Results from a large-scale study on data leakage in public notebooks.
- Recommendations on process designs that prevent data leakage.

We share our tool and supplementary materials on GitHub.[1]

2 OVERFITTING AND DATA LEAKAGE IN MACHINE LEARNING

Machine learning is the discipline of learning generalizable insights from data, typically in the form of a learned function, called *model*, that can make predictions for unseen data (e.g., production data). Developers building models with machine learning techniques usually follow an iterative and exploratory process [20] that is commonly depicted as a pipeline of multiple steps with feedback loops, including activities such as data collection, data cleaning, feature engineering, model training, model evaluation, and model deployment [2].

In model development, there is always the risk that the trained model *overfits* on the data used for training [37]—that is, it learns the patterns in the specific training data but generalizes poorly to unseen data. Therefore, it is customary to evaluate the accuracy of a model on data that was not previously used for training [37]—the evaluation measures to what degree the model predicts expected results for unseen data. For the evaluation to provide a meaningful approximation of the model's accuracy in production settings, the unseen data needs to be representative of the distribution of real data encountered in production.

Overfitting can happen whenever insight is gained from data, whether it is (a) a machine learning algorithm that is learning model parameters from data or (b) a human looking at data to make decisions about how to process the data or about what machine learning algorithm to use. Most importantly, due to the iterative nature of model development, it is common to evaluate different variants of a model to see whether accuracy improves with different decisions (e.g., different feature engineering, different machine-learning algorithm, different hyperparameters; some of this exploration may also be automated using AutoML approaches [11]). If decisions are based on prior evaluation, the data used in that evaluation influenced the training process and the model may overfit on it.

In summary, if we evaluate the model on data that was used in any form (automated or manually, directly or indirectly) in the socio-technical process used for training the model, the evaluation result may be overly optimistic because the model may have overfit on that data. In a technical sense, we want a *non-interference guarantee* in which the process of training the model is entirely independent of the data on which the model is evaluated.

Offline/Online Evaluation. The model evaluation we discuss above is usually executed offline *before* model deployment. Model developers could also conduct an online evaluation with production data *after* their model is deployed. Typically offline evaluations are conducted to gain confidence in the model before deployment and to avoid exposing users to low-quality models in production, just like software developers rely on unit testing to identify software bugs rather than only relying on crash reports and bug reports from users in production.

Training-Validation-Test Splits. In many settings, labeled data that can be used for training or evaluation is limited and expensive to gather. Many data science projects start with a single dataset, from which separate subsets are used for training and evaluation. The most common approach is to split data three ways into *training data*, *validation data*, and *test data*. Training data is used to develop the model and validation data is used for preliminary evaluation during model development (including hyperparameter tuning), whereas test data should just be used once as a final unbiased evaluation of the final model. Validation and test data seem similar and they are often used in the same type of evaluation functions in machine learning APIs, but they serve fundamentally different purposes—validation data is used for decision making during model development and hence not suitable for an independent evaluation.

The concepts of overfitting and the need to properly split data into these three sets to achieve unbiased evaluation results are universally covered in machine learning education and explained extensively in textbooks and course materials [e.g., 28, 37].

Data Leakage. Despite the conceptual requirement to never make any decisions that influence the model based on data that is used for evaluating the final model (i.e., noninterference of test data on model training), in practice, violations of this requirement are common and known as *data leakage* (because test data "leaks" into the training process) [4, 6, 19, 44]. We target three forms of data leakage:

- **Overlap Leakage:** An obvious form of leakage occurs when some or all test data is directly used as input for training or hyper-parameter tuning. More subtly, leakage can occur when creating training data based on test data in the form of data augmentation or oversampling, as in Fig. 2a. We call this type of leakage *overlap leakage*, as rows of test data overlap with rows of training data.
- **Multi-Test Leakage:** If data is used repeatedly for evaluation, it is highly likely that decisions are made based on that data, including algorithm selection, model selection, and hyperparameter tuning. For example, data scientists may have

[1] https://github.com/malusamayo/leakage-analysis

```python
# oversampling datasets, new rows are synthesized
    based on existing rows
X_new,y_new = SMOTE().fit_resample(X,y)
# splits after over-sampling no longer produce
    independent train/test data
X_train, X_test, y_train, y_test = train_test_split(
    X_new, y_new, test_size=0.2, random_state=42)

rf = RandomForestClassifier().fit(X_train,y_train)
rf.predict(X_test)
```

(a) Test data used for training

```python
# select the best model with repeated evaluation
results = []
for clf, name in (
        (DecisionTreeClassifier(), "Decision Tree"),
        (Perceptron(), "Perceptron")):
    clf.fit(X_train, y_train)
    pred = clf.predict(X_test)
    score = metrics.accuracy_score(y_test, pred)
    results.append(score, name)
```

(b) Test data used repeatedly for model selection

```python
# unknown words in test data leak into training data
wordsVectorizer = CountVectorizer().fit(text)
wordsVector = wordsVectorizer.transform(text)
invTransformer = TfidfTransformer().fit(wordsVector)
invFreqOfWords = invTransformer.transform(wordsVector)
X = pd.DataFrame(invFreqOfWords.toarray())

train, test, spamLabelTrain, spamLabelTest =
    train_test_split(X, y, test_size = 0.5)
predictAndReport(train = train, test = test)
```

(c) Test data leaked in preprocessing

Figure 2: Shortened data leakage examples from public notebooks.

selected the model that works best on the data. Data used repeatedly in evaluation, as in Figure 2b, can no longer be considered as unseen *test* data, but should be considered as *validation* data.

- **Preprocessing Leakage:** When training data and test data are preprocessed (transformed) together, test data sometimes influences the transformations of the training data. For example, data could be normalized according to the largest and smallest values in both training and test data, rather than only based on values from training data. Preprocessing leakage can occur in many transformations that consider multiple rows of the dataset, including feature selection (e.g., Fig. 1), normalizing data, projecting data with PCA, and vectorizing text data (e.g., Fig. 2c). In many practical settings, training and test data have very similar distributions and preprocessing leakage has only marginal influence on training data and hence the model; however, it is easy to construct examples where the mere knowledge about the distribution of test data can lead to substantial overfitting (see Fig. 1) and out-of-distribution predictions are particularly affected.

We target these forms of leakage because they are common sources of overconfident evaluation results, are discussed frequently both by practitioners and the literature [4, 44], and could be detected with static inspection of source code without understanding of the semantics of the data. Other forms of leakage are beyond the scope of this paper, including *label leakage* where unintended features in the data correlate with labels, leading to shortcut learning [4, 9, 19];

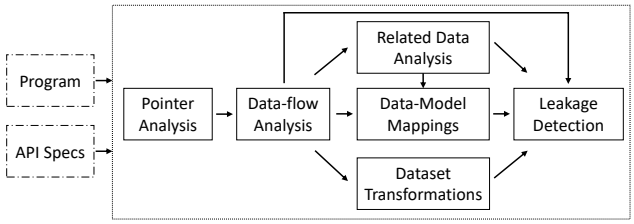


Figure 3: Approach Overview. Our analysis first performs standard pointer analysis and data-flow analysis, and collects domain-specific information (dataset transformations, related-data, data-model mappings) from the results. Finally, leakage is detected using all the information collected.

leakage from *incorrect splits* of data when dependencies between rows exist, such as in time-series data [4, 26]—both of these forms of leakage require a deep understanding of the semantics of the data and are orthogonal to the three forms of data leakage we address.

3 APPROACH

We developed a static code analysis approach to detect different forms of data leakage. We analyze how data flows through notebook code and how it is used for training, validation, and testing. To this end, we statically collect specific information needed to detect leakage (see Fig. 3):

- **Dataset transformations.** In the preprocessing steps, transformations may leak information across rows, duplicate rows, or transform rows independently. Some of these transformations could contribute to preprocessing leakage or overlap leakage. Therefore, we need to track how datasets are processed. To this end, we label data-flow edges to track different kinds of dataset transformations.
- **Related-data relations.** To map data to models and detect overlap leakage, we need to understand whether datasets may have originated from the same rows in an original dataset. We track this with a related-data *relation* on top of our standard data-flow relations that tracks which two variables are related.
- **Data-model mappings**. To detect leakage, we need to identify the training/validation/test data for a given model. Here the key challenge is to differentiate validation and test data. We collect this information in data-model mappings, built on top of the related-data relations.

Once we have collected data flows (including data transformations and related-data relations) and data-model mappings, we can detect leakage by matching patterns over this information. In the remainder of this section, we explain each of these steps using a running example (Fig. 4).

3.1 Tracking Data Flows

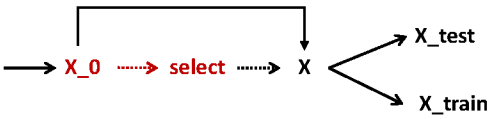
For all leakage detection, we need to identify how datasets and other computations relate to each other in the notebook. That is, we need to track how data may be repeatedly transformed and split, possibly by using information originally derived from other parts of the data, until it flows into training or evaluation functions of models. To this

```
1  import pandas as pd
2  from sklearn.feature_selection import SelectPercentile
       , chi2
3  from sklearn.model_selection import LinearRegression,
       Ridge
4
5  X_0, y = load_data()
6
7  select = SelectPercentile(chi2, percentile=50)
8  select.fit(X_0)
9  X = select.transform(X_0)
10
11 X_train, y_train, X_test, y_test = train_test_split(X,
       y)
12 lr = LinearRegression()
13 lr.fit(X_train, y_train)
14 lr_score = lr.score(X_test, y_test)
15
16 ridge = Ridge()
17 ridge.fit(X, y)
18 ridge_score = ridge.score(X_test, y_test)
19
20 final_model = lr if lr_score > ridge_score else ridge
```

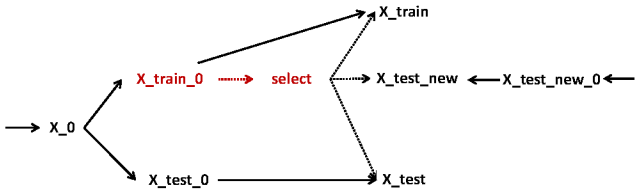
(a) The code here contains all three kinds of train-test leakage. In line 9-10, there is a pre-processing leakage, where it selects features based on both training data and test data. In line 17-18, there is an overlap leakage, where test data is used for training. There is also a multi-test leakage, since *X_test* is evaluated multiple times and there is no other independent test data. Leakage-prone operations (reduce edges) and train/test locations are highlighted in red/yellow.

```
21 import pandas as pd
22 from sklearn.feature_selection import SelectPercentile
       , chi2
23 from sklearn.model_selection import LinearRegression,
       Ridge
24
25 X_0, y = load_data()
26
27 X_train_0, y_train, X_test_0, y_test =
       train_test_split(X_0, y)
28
29 select = SelectPercentile(chi2, percentile=50)
30 select.fit(X_train_0)
31 X_train = select.transform(X_train_0)
32 X_test = select.transform(X_test_0)
33
34 lr = LinearRegression()
35 lr.fit(X_train, y_train)
36 lr_score = lr.score(X_test, y_test)
37
38 ridge = Ridge()
39 ridge.fit(X_train, y_train)
40 ridge_score = ridge.score(X_test, y_test)
41
42 final_model = lr if lr_score > ridge_score else ridge
43 X_test_new_0, y_test_new = load_test_data()
44 X_test_new = select.transform(X_test_new_0)
45 final_model.score(X_test_new, y_test_new)
```

(b) We could fix the code above by moving feature selection later (after train/test splits), use only training data for training *ridge*, and use an independent test data for evaluation in the end.



(c) Data-flow graph for the code. Only important variables are shown. Edges between datasets are solid, while other edges are dotted. Reduce edges, together with their variables, are marked in red.



(d) Data-flow graph for the code after fix. *X_train_0* is no longer related to *X_test*, eliminating the pre-processing leakage. With changes to training data and the introduction of a new test set, the overlap leakage and multi-test leakage are also fixed.

Figure 4: Example code with data-flow graph (before and after fix) for leakage analysis.

end, we perform standard data-flow analysis through assignments and method calls. In addition, we collect additional information about dataset transformations and related-data relations:

3.1.1 Dataset transformations. When tracking data flow, we are particularly looking for flows that are prone to leak information across multiple rows in a dataset or create new rows from old rows. The key insight is that transformations that remove rows or process data one row at a time are not problematic from a leakage perspective, but transformations that process multiple rows together (as simple as counting or as complex as normalizing data by the largest value in a column) may cause leakage as rows may no longer be independent. In terms of typical concepts in functional programming, *map*-like and *filter*-like[2] transformations are okay, but *reduce*-like transformations are suspicious.

To this end, our analysis labels (a) data-flow edges between datasets that perform computations independently on each row (map-like transformations) as *map* edges and (b) edges originating from datasets that perform computations across rows (reduce-like transformations) as *reduce* edges. We manually label APIs in popular libraries (numpy, pandas, and sklearn) to identify those that perform computations per rows, such as *fillna* (replaces missing values with the provided value) and those across multiple rows, such as *numpy.mean* (computes a value based on multiple rows) and *SelectPercentile.fit* (learns features from analyzing all rows). Notice that a *reduce* edge may produce a single scalar value (such as

[2] Technically, filter functions may change the length of a dataset and could leak information in the sense of non-interference. Pragmatically, we consider filter as unproblematic for data leakage, as filter is commonly used by data scientists for removing data points and therefore reporting this would not be very useful.

Inputs

$\mathbf{V}$	program variables
$\mathbf{D} \subseteq \mathbf{V}$	datasets
$\mathbf{M} \subseteq \mathbf{V}$	models
$DataFlow \subseteq \mathbf{V} \times \mathbf{V}$	data flow paths (transitive closure)
$DatasetFlow \subseteq \mathbf{D} \times \mathbf{D}$	data flow paths between datasets
$MapEdge \subseteq \mathbf{D} \times \mathbf{D}$	map-like operations
$ReduceEdge \subseteq \mathbf{D} \times \mathbf{V}$	reduce-like operations
$DupEdge \subseteq \mathbf{D} \times \mathbf{D}$	duplication operations
$ModelData \subseteq \mathbf{M} \times \mathbf{D} \times \mathcal{P}(\mathbf{D}) \times \mathcal{P}(\mathbf{D})$	models with corresponding training, validation, and test datasets

Rules

$$\frac{a \in \mathbf{D} \quad b \in \mathbf{D} \quad (a, b) \in DatasetFlow}{(a, b) \in RelData} \text{ reldata/flow}$$

$$\frac{(b, a) \in RelData}{(a, b) \in RelData} \text{ reldata/sym} \qquad \frac{a \in \mathbf{D}}{(a, a) \in RelData} \text{ reldata/ref}$$

$$\frac{(a, b) \in RelData \quad (a, c) \in RelData \quad (a, c) \in MapEdge}{(b, c) \in RelData} \text{ reldata/map}$$

$$\frac{(src, a) \in DupEdge \quad (a, b) \in RelData \quad (a, c) \in RelData}{(b, c) \in RelData} \text{ reldata/dup}$$

$$\frac{\begin{array}{c}\exists (m, d_{tr}, D_{va}, D_{te}) \in ModelData,\ D_{va} \neq \emptyset \\ \forall (m, d_{tr}, D_{va}, D_{te}) \in ModelData,\ D_{te} = \emptyset\end{array}}{\textit{multi-test leakage in notebook}} \text{ leak/multi}$$

$$\frac{\begin{array}{c}(m, d_{tr}, D_{va}, D_{te}) \in ModelData \\ \forall d_e \in D_{te} \cup D_{va}, (d_e, d_{tr}) \in RelData\end{array}}{\textit{leakage due to overlap between } d_{tr} \textit{ and } D_{te} \cup D_{va} \textit{ for model } m} \text{ leak/overlap}$$

$$\frac{\begin{array}{c}(m, d_{tr}, D_{va}, D_{te}) \in ModelData \quad (s, t) \in ReduceEdge \\ d_e \in D_{te} \cup D_{va} \quad (d_e, s) \in RelData \quad (t, d_{tr}) \in DataFlow\end{array}}{\textit{leakage between } d_{tr} \textit{ and } d_e \textit{ through the transformation from } s \textit{ to } t} \text{ leak/pre}$$

Figure 5: Notations and rules for leakage detection from input relations.

calculating the mean of a column) which may be used subsequently in a map-like transformation (e.g., replacing missing values with that mean). Our analysis will later use this information to detect leaks when data flows through a *reduce* edge.

In a similar fashion, we also identify transformations that duplicate rows (e.g., *SMOTE.fit_resample* from Fig. 2a) as *duplicate* edges, which are potentially problematic because they can create dependencies between rows within a dataset, making random splits no longer produce independent datasets.

In our running example (Fig. 4a), we see a preprocessing operation that may introduce leaks: *SelectPercentile.fit*, as it uses the distribution information of the input data. It corresponds to a *reduce* edge from *X_0* to *select*. Operation *SelectPercentile.transform*, on the other hand, corresponds to a *map* edge from *X_0* to *X*.

3.1.2 Related-data relations. In many steps for detecting leakage, we need to identify whether two datasets are related or independent. For example, to reason about repeated evaluations with test data (multi-test leakage), we need to understand whether the data used in two test locations are independent or somehow related. Technically, we establish a *related-data relationship* to track whether two variables relate to each other (*RelData* in Fig. 5). We consider two

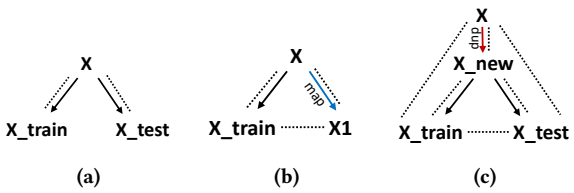


Figure 6: Related-data relations are usually not transitive (a), except for duplicate edges (b) or map edges (c). Solid arrow edges are normal data-flow edges; dotted edges are related-data relations.

datasets as related (a) if they contain the same data, (b) if one is derived from the other, or (c) if they share rows from the same origin.

First, two datasets with a direct dataset flow (i.e., data-flow that only considers datasets) relation are considered as related (rule *reldata/flow* in Fig. 5). This is commonly the case when one dataset is a transformed version of another (e.g., Line 9 in our running example transforms *X_0* into *X*) or simply assigned from one variable to another. Also, splitting a dataset creates multiple datasets that are each considered related to the original dataset (e.g., in Line 11 in our running example, *X_train* and *X_test* are both related to *X*). Our related-data relationship is reflexive and symmetric (rules *reldata/ref* and *reldata/sym* in Fig. 5).

When two datasets are derived from the same original dataset, reasoning about their relationship is more complicated. In the common case that a dataset is split correctly into independent training and test sets, these two derived datasets should not be considered as related. However, when two datasets are derived from the same rows of an original source (e.g., two different ways of normalizing the entire dataset) they should be considered as related. Ideally, we would reason about transformations at the row level and consider datasets as related if their rows' origins in another dataset overlap, but corresponding dynamic analyses [14, 16, 25] create runtime overhead and are difficult to adapt to our analysis. Instead, we approximate the most common patterns statically.

By default, we assume that two datasets derived from the same origin dataset are derived with a correct split and are hence considered as not related (Fig 6a). However, if we know that one of the datasets is derived with a map-like transformation (*map edge* in Sec 3.1.1) that dataset shares all rows with the original dataset and is not involved in a split. Hence, we consider that dataset as related to all other datasets derived from the same origin (Fig. 6b; rule *reldata/map* in Fig. 5). Furthermore, if the origin dataset contains dependent rows (e.g., from a data augmentation step), we assume that derived datasets are also related, as splits are unlikely to maintain independence. Hence, we consider all datasets derived from an origin where data was produced (directly or indirectly) through a duplicate edge (see Sec 3.1.1) as related (Fig 6c; rule *reldata/dup* in Fig. 5).

Note that our three heuristics approximate a more accurate dynamic analysis, but cannot cover all cases. For example, our heuristic would identify a split with overlapping rows (*a, b = o[:100], o[50:]*) as independent, but those mistakes are very rare. We consider that the cost of increased false positives or more expensive dynamic analysis outweigh the benefits of detecting such very rare cases. As we will show in our evaluation in Section 4.2.3, our heuristics

capture the relationships found in common notebooks to achieve relatively accurate leakage detection.

3.2 Data-Model Mappings

While data-flow analysis tracks how data is transformed and moves through the notebook, we can only identify whether data is training data, validation data, or test data by determining how it is eventually used.

As a first step, we identify program locations where (usually preprocessed) data is used for training, validation, and testing. We identify those locations simply by finding API calls in the notebook that are typically used for training and evaluation purposes, such as the *fit* and *predict* functions in *sklearn*'s APIs. We later trace back the source of the data used in these APIs to processing steps and original datasets using standard data-flow analysis.

While training data can be identified with distinct function calls, distinguishing between validation and test data is conceptually challenging because both are used with the same APIs (e.g., predict). We cannot reliably infer whether a data scientist intends to use data for validation or testing purely based on notebook code—this is a challenge even for human experts who may need to rely on context clues or documentation. We therefore rely on a simple heuristic that considers data that is used repeatedly in evaluation as validation data and all data that is used only once in evaluation as test data. We consider data to be evaluated repeatedly if its location is within a loop or if two locations are connected to evaluate the same or related data as per our data-flow analysis (i.e., variables in both locations are connected through the related-data relation from Sec. 3.1.2).

Finally, we group each training dataset with the corresponding validation and test datasets that relate to the same model. We use standard data-flow analysis to identify which call locations share the same target object. To account for possible repeated training of the same model object, we always group training data with all subsequent validation and test data, until the next training data is identified. In the end, we derive a series of model-data tuples (*ModelData* in Fig. 5), where the same training data might correspond to zero or multiple validation/test datasets. In our example (Fig. 4a), we show two model-data tuples based on this heuristic: (*lr*, *X_train*, {*X_test*}, ∅) and (*ridge*, *X*, {*X_test*}, ∅).

3.3 Leakage Detection

After collecting the above information, identifying leakage is performed through pattern matching:

- To detect multi-test leakage, we check for a given model whether there exists at least one piece of test data. Note that 'test data' evaluated multiple times will already be identified as validation data in our analysis (see Sec. 3.2). If for all models, there is no test data but only validation data detected, we report multi-test leakage (rule *leak/multi* in Fig. 5).
- To detect overlap leakage, we check for a given model whether training data and test/validation data[3] are related. Note that there might be multiple test/validation data in a single model-data tuple (see Sec. 3.2 for how we derive the tuple). Therefore, for a given trained model, we only report overlap leakage when all of its test/validation data overlaps with the training data (rule *leak/overlap* in Fig. 5).
- To detect preprocessing leakage, we check whether training data contains information from test/validation data through preprocessing (*reduce* edges). If training data uses reduced information from test/validation data (or datasets that are related to test/validation data), we will report a case of preprocessing leakage (rule *leak/pre* in Fig. 5).

In our example, we could find a path from *X* to *X_train* and a path from *X_0* to *X_test* (see Fig. 4c). As *X* contains reduced information from *X_0*, which is related to *X_test*, we establish that test data information is leaked into training data, and there is a preprocessing leakage. Next, because *X* is related to *X_test* (as *X_test* is transformed from *X*), there is an overlap leakage when we evaluate the second model *ridge*. Finally, the two trained models share the same test data (*X_test*), which we will identify as validation data. Because there is no independent test data used in the final evaluation, we conclude that there is also a multi-test leakage.

In the fixed version of our example (see Fig. 4b), we see that the two model-data tuples are changed to (*lr*, *X_train*, {*X_test*}, {*X_test_new*}) and (*ridge*, *X*, {*X_test*}, {*X_test_new*}) and hence: (1) models no longer contain information from *X_test*, as the reduced information only comes from *X_train* (see Fig. 4d)), eliminating the preprocessing leakage, (2) in all tuples, training data and test/validation data are no longer related, eliminating the overlap leakage, and (3) there is independent test data *X_test_new* that is evaluated only once, eliminating the multi-test leakage.

3.4 Implementation

To make our analysis easy to extend and modify, our implementation uses *datalog*, a language commonly used in declarative program analysis [3, 42]. Our two-phase implementation first transforms Python code into datalog facts as an intermediate representation and then analyzes these facts to generate leakage detection results. Our analysis design is similar to *doop* [3], a popular Java program analysis framework.

In the front end, we generate datalog facts that can be easily analyzed subsequently. Specifically, we translate complex language structures into simpler ones, translate assignments (both variables and fields) to static single assignment form, which ensures that subsequent analyses are flow-sensitive, and match method invocations with signatures. To identify datasets and targets of method invocations, we perform type inference with the off-the-shelf type inference engine *pyright* [27].

Based on initial datalog facts, we compute additional facts for the relations (e.g., *RelData*, *ModelData*) described above and subsequently detect data leakage using datalog queries. We implement a standard Anderson-style 2-call-site-sensitive pointer analysis similar to *doop*, with special treatment of common language features (e.g., lists and global variables). The data-flow analysis is built on pointer analysis and also follows standard implementations.

Our analysis requires specifications of data science APIs. Specifications are mainly used to provide domain knowledge (e.g., which APIs are used for training/testing, which APIs behave *reduce*-like).

[3] Note that for overlap/preprocessing leakage, we do not distinguish between test and validation data. Leakage between training data and validation data is still problematic, as it defeats the purpose of validation data (i.e., providing independent validation during model development).

Our current implementation supports three machine learning libraries – *sklearn*, *keras*, and *pytorch* – and two libraries commonly used for data transformations – *pandas* and *numpy*. We went through the official documentation of these libraries to find APIs that perform data transformations and APIs that perform supervised learning. Our analysis can be easily extended to other libraries by providing their specifications.

4 EVALUATION

We first evaluate the accuracy of our analysis, that is, its ability to find actual leakage and to avoid false alarms:

- **RQ1**: *How accurate are the results of our analysis?*

To ensure that our analysis can be used in an interactive or continuous integration setting during model development, we also evaluate efficiency in terms of running time:

- **RQ2**: *How efficient is our analysis?*

Finally, after establishing accuracy and efficiency, we use our analysis to study test-train leakage in a large corpus of notebooks, exploring common forms and sources of leakage. We will show that leakage is common across different types of notebooks. In addition, leakage often manifests itself in nontrivial data flows in notebooks, in forms that can be tedious or even difficult to detect manually, providing strong, albeit indirect evidence for the usefulness of our automated detection:

- **RQ3**: *How prevalent is data leakage in public notebooks?*
- **RQ4**: *What do typical leakage issues look like?*

4.1 Research Design

We evaluate all four research questions with a corpus of public data science code in Jupyter notebooks. For different research questions, we use different subsets of this corpus.

Corpus of notebooks with data science code. To answer our research questions, we curate a large corpus of public notebooks with data science code. Specifically, we collect Jupyter notebooks from GitHub and Kaggle. GitHub is a common platform for storing data science code for a range of purposes, from hobby and educational projects, to research projects and tutorials, to production systems. Kaggle is a common platform for data science competitions where users can submit notebooks as solutions to competition problems. We purposely selected code in notebooks, rather than arbitrary Python files, because notebooks are the primary environment for developing data science code [32].

For GitHub, we collected *all* notebooks from GitHub repositories created in September 2021 (strictly independent from all notebooks from earlier periods used during development of our analysis). Specifically, we used the GitHub search API to identify repositories with notebook code and partitioned the search space to collect *all* 81,026 repositories. By selecting all notebooks from a recent time period, we get a full and representative sample of the different kinds of notebooks published on GitHub. We collected a total of 280,994 notebooks this way.

For Kaggle, we selected a smaller and more targeted notebook population, collecting notebooks from two popular competitions, *titanic* and *housing* [17, 18]. For each competition, we collect the 200 notebooks with the most votes and the 200 most recent notebooks, as of April 12, 2022. We selected these competitions, because they use tabular data and require significant preprocessing effort. We use this corpus to understand leakage issues among typical competition solutions that are prone to data leakage. They are not necessarily representative of all competition solutions on Kaggle.

We further filter these notebooks to include only those that use the machine learning libraries supported by our current implementation (*sklearn*, *keras*, and *pytorch*). The discarded notebooks either only use not-yet-supported libraries such as *tensorflow* or do not train any models. This leaves us with 107,603 GitHub notebooks and 108,273 in our corpus overall. In Table 1, we show descriptive statistics of our final corpus.

Analyzing accuracy (RQ1). Establishing ground truth for data leakage is challenging and we are not aware of existing datasets. To evaluate the accuracy of our analysis, we measure both false positives and false negatives on a sample of notebooks for which we manually establish ground truth. Due to the substantial manual effort involved, we perform the analysis on 100 randomly sampled notebooks from our GitHub corpus (which yields an 8 % error margin with a confidence level of 95 %) [23].

One author manually labeled these 100 notebooks looking for leakage issues and then compared manual labels with the analysis results. For all notebooks where the manual labels and analysis results disagreed, the author sought the expert opinion of a second author (a trained data scientist). Together they discussed the issue to determine whether the notebook contained leakage, correcting the ground truth label if needed. This correction was needed in 7 notebooks, which were incorrectly labeled initially, arguably indicating that manual checking of data leakage is non-trivial and error-prone, even for experts. Overall our process balances labeling effort and confidence in the ground truth.

As per our manually established ground truth, 40 of the 100 notebooks contained at least one form of leakage – 20 with preprocessing leakage, 8 with overlap leakage, and 32 with multi-test leakage.

Analyzing efficiency (RQ2). For efficiency, we recorded the execution time of our analysis for *all* notebooks in our GitHub corpus. We record timing separately for the analysis front end (collecting facts), the type inference, and the actual analysis (datalog engine). We set a timeout of 5 minutes per notebook. The experiment is conducted on a Precision 3650 workstation, with Intel(R) Xeon(R) W-1350 CPU and 32GB memory. The time is measured using Python's time module (wallclock time).

Analyzing leakage frequency (RQ3). We analyze leakage for the entire corpus and report the frequency with which we raise warnings for each kind of leakage. Note that we consider at most one warning per leakage kind per notebook to avoid biasing results with some notebooks that raise lots of warnings.

To further understand whether leakage associates with certain types of notebooks, we separately report leakage for different subpopulations. Specifically, we break down results for the following subpopulations:

- *Popular notebooks* are viewed (and possibly reused) by more people, thus having more potential to spread problematic practices. We conjecture that popular notebooks come from more experienced data scientists and are better crafted. We

Dataset	#notebooks	LoC	#stars	Preprocessing Leakage	Overlap Leakage	Multi-test Leakage	Any Leakage
GitHub (all)	107,603	410	1.1	12.3	6.5	18.5	29.6
GitHub (popular)	920	378	95.2	4.9	5.2	15.9	20.9
GitHub (tutorials)	1,157	584	4.2	3.9	2.9	11.3	16.2
GitHub (assignments)	7,576	559	0.6	13.9	7.4	22.0	33.0
Kaggle (top)	312	851	-	56.1	N/A	N/A	N/A
Kaggle (recent)	358	504	-	55.8	N/A	N/A	N/A

Table 1: Data Leakage Distribution. LoC is the average number of lines of code across all notebooks in the group. Number of stars is based on the repository that these notebooks reside in and is also averaged. We show the percentage of notebooks for which we report each leakage type. For Kaggle notebooks, we only track preprocessing leakage because the other two are infeasible in this setting (marked as N/A in the table).

identify 920 such popular notebooks as those in GitHub repositories in our corpus with 10 or more stars.

- *Tutorial notebooks* similarly are explicitly designed for teaching others and could spread problematic practices. We identify 1,157 tutorial notebooks by searching for the phrase *"this tutorial"*.
- *Assignment notebooks* contain solutions to course projects and assignments. We conjecture that these notebooks better represent practices of beginners than average notebooks in our corpus. We identify 7,576 assignment notebooks on GitHub by searching for the keywords *'homework'* and *'assignment.'*
- *Competition notebooks (popular and recent)* are written by a mix of experienced and learning data scientists, possibly with an increased incentive to maximize model accuracy. Here, we report results from the Kaggle notebooks from our corpus.

Analyzing leakage characteristics (RQ4). We measure distance between different program constructs by measuring lines of code between them (based on the Python files converted from notebooks using *nbconvert* in the default setting) and compare them to the length of the entire notebook. We expect that issues that span longer distances are harder to analyze manually. Specifically, we calculate distance between leakage sources (e.g., reduced data) and training locations for preprocessing leakage, and distance between different evaluation locations for multi-test leakage. We report the results for the entire GitHub corpus.

For the whole dataset and each sub-population, we explore the distributions of different leakage issues, complexities of these issues, and also how they are distributed across different sub-populations.

Threats to validity. Establishing accurate ground truth for leakage is challenging. Our experience shows that even data science experts looking for leakage may miss it in complex data flows. We adopt a best effort approach with human labeling and comparisons with automated results that balance effort with confidence. We share our data for independent validation.

For RQ3 and RQ4, we report leakage warnings but validating all warnings is simply infeasible at this scale. Our results should therefore be interpreted with the error margins established in RQ1. In addition, warnings about multi-test leakage may be rooted in settings where independent test data may exist outside of the notebook; our evaluation would also not detect multi-test leakage if the same test data was used repeatedly in past versions of a notebook or is used repeatedly in multiple notebooks. Generally, our analysis provides only a piece of a larger picture that needs to involve process design and other assurances, as we will discuss in Section 5.

The notebook population in our corpus is representative of public notebooks on GitHub (and some Kaggle competitions), but may not generalize to data science code outside of notebooks, across multiple notebooks, or to proprietary data science pipelines. Readers should hence be careful when generalizing our results.

Finally, our analysis focuses on the presence of leakage, not whether data scientists find leakage reports actionable or what effect leakage has in overestimating the reported accuracy results. We leave such evaluations to future work but point again to the fact that leakage is firmly established as problematic in educational material (cf. Sections 1–2).

4.2 Results

4.2.1 Analysis accuracy (RQ1). Our analysis prototype successfully executed for 94 of the 100 notebooks in our labeled RQ1 sample; the remaining 6 notebooks failed due to syntax errors. For the 94 notebooks, our prototype found 15 with preprocessing leakage, 7 with overlap leakage, and 19 with multi-test leakage. All the found leakage issues were true positives except for one case due to over-approximation in related-data analysis, yielding a precision of 97.6% (40 out of 41 detected issues).

On the other hand, our prototype missed 19 issues due to unsupported libraries (6), mistaking single test cases as test data (3), storing/loading model in external storage (3), undetected test data evaluation (3), inaccurate type inference (2), and under/over-approximation in related-data analysis (2). This yields a recall of 67.8% (40 out of 59).

Overall, our prototype achieves an accuracy of 92.9% (262 out of 282 potential leakages). Analysis for preprocessing/overlap leakage is more accurate than multi-test leakage in this sample. Based on these results, we conclude that our analysis is generally accurate.

4.2.2 Analysis overhead (RQ2). Most (92.78 %) of the 107,603 GitHub notebooks in our corpus could be analyzed successfully within the 5 minute time limit. On average, our analysis completes within 3.23 seconds, with most of the time (2.20 seconds on average) spent on type inference. A small percentage of notebooks could not be analyzed due to syntax errors (7.08 %), timeout (0.09 %; usually due to explosion from context-sensitivity in our data-flow analysis), and language features not supported by the front-end parser (0.05 %, e.g., named expression introduced in PEP 572). *We conclude that*

our analysis is efficient enough for interactive use and in continuous integration settings.

4.2.3 Leakage in public notebooks (RQ3). Our analysis reports at least one form of leakage for almost a third of all public notebooks in our GitHub corpus (29.6%, see Table 1). *We found frequent evidence of all three forms of leakage.*

Preprocessing leakage is prevalent in notebooks. Overall, our analysis reported preprocessing leakage for 12.3% of notebooks in our corpus. The most common sources of leakage during preprocessing are scaling, computing mean and standard deviation, and using results of a principal component analysis (PCA) in downstream data transformations. For text data, the most common source of preprocessing leakage is vectorizing through counting or *tf-idf* over the whole dataset. Zooming in, we reported preprocessing leakage in 32.9% of those notebooks that scale their data, 8.4% of those that compute mean or standard deviation, and 13.4% of those that perform PCA.

Many notebooks lack independent test data. We report multi-test leakage in 18.5% of all notebooks in our corpus. We also detected that among all notebooks that train a model, 53.9% contain validation data (i.e., data that is used repeatedly for evaluation), but 35.0% of trained models are not evaluated with *independent* test data. Models evaluated with validation but without test data represent 28.3% of all trained models in our GitHub corpus.

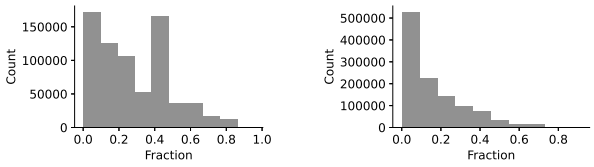
Training data often overlaps with validation and test data. We report overlap leakage in 6.5% of all notebooks in our corpus. A closer look reveals that 8.0% of all trained models are only evaluated on data that overlaps with its training data.

Leakage is common in both beginner and expert code. When analyzing the subsets of our corpus, we report leakage in all subsets, but to different degrees (see Table 1). Assignment notebooks more representative of beginners are more likely to receive reports for all leakage types. In contrast, popular notebooks and tutorials most likely associated with more advanced data scientists are slightly less likely to receive leakage reports, but leakage is still reported fairly commonly (20.9 % of popular and 16.2 % of tutorial notebooks). The fact that leakage seems common even in tutorial notebooks designed for educational purposes seems concerning.

Finally, the rate of reports of preprocessing leakage is very high in competition notebooks (>55 %). Indeed, it is not uncommon that competitors concatenate separately provided train and test data before preprocessing the combined dataset. The way the competitions are designed (providing values but not labels of test data) may encourage exploiting leakage to maximize accuracy results, even, or especially by, experts. At the same time, because test labels are not provided, we do not report overlap and multi-test leakage. We will discuss competitions separately in Section 5.2.2.

4.2.4 Leakage characteristics (RQ4). We observe that leakage often occurs in patterns that make it challenging to detect manually.

Leakage issues exhibit non-local patterns. For preprocessing leakage, the average distance between the leakage source (the *reduce* edge) and the location where the training data is used is 293 lines of code. In more than half of the cases, the distance between leakage source and training location is more than 20% of the length of the notebook (see Fig. 7a). This distance illustrates the often long processing sequences and non-local data flows that are difficult to analyze manually.



(a) Distribution of fraction of distance between preprocessing leak sources and training data.

(b) Distribution of fraction of distance between test locations that evaluate the same data.

Figure 7: Leakage issues exhibit non-local pattern.

For multi-test leakage, the average distance between two locations evaluating models with the same (or related) data is 255 lines of code. In more than 30% of all cases, test location distance is more than 20% of the whole notebook (see Fig. 7b). On average, there are 4.4 model-evaluation locations that use the same (or related) test data in notebooks with a multi-test leakage warning. This similarly illustrates the non-local reasoning required to notice this form of leakage.

A single notebook often trains multiple models. Among all notebooks in our GitHub corpus, 65.3% train at least one model (in sklearn, pytorch, or keras) and 66.0% evaluate at least one model (5.8% of notebooks do not train but evaluate a model, typically when loading a pre-trained model). Among the notebooks that do train at least one model, we found that 54.3% train multiple models. Having to commonly track multiple models and how data flows into their training and evaluation can be another challenge when manually reasoning about leakage.

5 DISCUSSION

Our results indicate that static detection of several forms of data leakage is feasible and that this kind of leakage is pervasive in practice. At the same time, it is not a comprehensive solution to avoid leakage or other forms of overfitting.

5.1 Practical Impact of Data Leakage

Not all leakage issues are equally problematic and some data scientists developing models may consider that some forms of leakage (e.g., the median of a column) are entirely negligible and that therefore warnings about leakage are not actionable and hence *effective false positives* [38]. We even found some tutorials where the description explicitly indicates that developers are aware of leakage problems but ignore them anyway, such as testing with part of the training data "for the sake of simplicity".

We have seen and heard of a large range of different impacts. On the one hand, we have heard (personal communication) of multiple cases where models were accidentally evaluated on training data producing entirely misleading results in research teams at BigTech companies, and that it is easy to create artificial examples where preprocessing leakage creates substantially inflated accuracy results (e.g., Fig. 1). On the other hand, experiments with notebooks in our corpus often just yielded marginal if any differences in accuracy.[4]

[4]Due to the known problems of reproducing public notebooks [34, 46], we were not able to perform systematic experiments on a larger sample.

We expect that more substantial leakage, such as evaluating on the training data mostly stems from simple mistakes (such as using the wrong variable name) – which *practitioners* can easily detect with our analysis. The more subtle leakage through preprocessing may often have little effect on reported accuracy in most cases, but we still argue that it represents a bad coding pattern. In our evaluation, we found that even many tutorials and top competition solutions leak their test data, let alone homework assignments. We think that in particular *educators* should insist on avoiding leakage in their course materials and homework, and our analysis provides an easy way to create awareness of the most common patterns leading to leakage.[5]

5.2 Process Design for Preventing Data Leakage

While we intend our analyzer to be used primarily as an educational tool and a tool to surface common mistakes in practice (either directly in notebook environments or integrated into code reviews [38]), a more robust solution can be achieved by designing the process of how responsibilities are assigned. This is particularly relevant in formal settings where model development is outsourced [29] and in data science competitions.

5.2.1 Contract settings. In settings where a team is given data to develop a model as part of a contract (e.g., outsourcing), data leakage can systematically be avoided if test data is not provided to the data scientists who develop the model in the first place, but instead reserved for external evaluation upon delivery. When data scientists do not have access to test data, they cannot derive any insights from it, cannot use it repeatedly in evaluations, and cannot even manually look at data distributions to inform modeling decisions—even in settings where a data science team would have an incentive to cheat to present inflated accuracy numbers. The drawback of such a process design is that an additional external evaluator is needed who must have enough understanding of the data to be able to split it appropriately into training and test data (which can be nontrivial when dependencies exist [10, 26, 39, 43]). The evaluator also needs to have access to the model to run it locally, to not risk leaking test data during model inference. Also, the evaluator cannot repeatedly report results from the evaluation back to the developers of the model. Notice that test data does not become immediately useless, but gradually loses confidence which can be accounted for in scores, as explored in detail elsewhere [7, 35].

In many practical settings though, the team developing the model is involved in acquiring or collecting the data in the first place [29]. In such settings, it would be paramount for an external evaluator to independently collect data, which is often infeasible or prohibitively costly, as it might require replicating the expertise gained by the model development team during the development process. If there is some trust relationship between model developers and model users, approaches that foster best practices and avoid common mistakes, such as our static analysis tool, might be a better pragmatic alternative.

[5]We see some practitioners and educators agree with this stance. For example, in this GitHub issue (https://github.com/keras-team/keras/issues/1753), participants actively discuss the potential problems of tutorials setting a bad example for learners, even when there technically is no actual leakage problem in the specific example.

5.2.2 Data science competitions. Competitions often pursue a similar form of external evaluation, but often make compromises to better automate the competition, reduce cost and complexity, or increase engagement with more granular feedback:

Withholding test labels. A common design for Kaggle competitions is to provide test data without labels. The competitors submit the predicted labels to receive a test score. The operational advantage of this approach is that the competition organizer does not need to execute the submitted models—it avoids (1) executing submitted (untrusted) code, (2) having to support a range of different models, and (3) bearing the cost of model inference during evaluation. While this design prevents overlap leakage and multi-test leakage, preprocessing leakage is still possible. In fact, our evaluation of the two Kaggle competitions, which both use this design, shows that competitors often exploit preprocessing leakage.

Limiting repeated submissions. Most competitions allow participants to submit multiple revisions of a solution, receiving scores for each of them, risking multi-test leakage. To minimize this risk, some competitions limit the number of submissions per team. However, if participants can see other solutions in a leaderboard or create multiple accounts, leakage is still a concern even if the actual test data is withheld from competitors.

Partial test scores. Some competitions split their hidden test data and provide feedback on incremental submissions only with part of the test data, until finally scoring all submissions exactly once at the end of the competition with the remainder of yet unused test data. This effectively separates the hidden evaluation data into validation and test data. It still allows to provide meaningful preliminary feedback on a leaderboard during the competition, without allowing overfitting in the final results. As a downside, competitors that are better at extracting insights from partial test scores may have an advantage in the competition over those that just work with the training data, hence encouraging explicit attempts to leak information.

While a once-only external evaluation with hidden test data at the end of the competition may be the cleanest solution, in practice competition designers will often want to make compromises with one of the designs above, which each reduce leakage to some extent but cannot avoid it entirely. Adding our static analyzer to the competition infrastructure could call out bad practices or could remind observers to not use the same practices outside competitions.

5.3 API Design for Preventing Data Leakage

It is also possible to avoid certain leakage issues through API design. For example, many ML libraries advocate the use of pipelines for preprocessing [40]. If correctly used, they could help avoid preprocessing leakage, because the library enforces that only training data is used to extract parameters for preprocessing.

From our GitHub corpus, we observe that only 5.5% notebooks use pipelines. Surprisingly, 18.1% of these notebooks still contain pre-processing leakage. When we inspected samples from these notebooks, we found that they often do not apply pipelines to the whole preprocessing stage, and some of them even use pipelines in the wrong way. This informs us that better education on how to use these APIs is as important as designing these leakage-proof APIs themselves.

5.4 Limitations and Alternatives

Our definitions of leakage and corresponding analyses are limited to the scope of what is observable statically in data science code (typically in a notebook). In addition, our static approximation relies on models of library functions and several heuristics.

Importantly, our approach cannot detect repeated evaluations that are not present in the notebook (e.g., cells modified and evaluated repeatedly) and cannot detect test data outside the notebook. It may therefore issue spurious multi-test leakage warnings or miss some. Our evaluation also shows that our analysis misses some leakage but produces few false positives (recall: 67.8%, precision 97.6%), therefore the reported leakage may be underestimated. Similar to many static analysis warnings, we envision warnings as a starting point for reflection and discussion (e.g., during code review [38]), and not necessarily as a blocking issue that always needs to be addressed with code changes. For example, developers might compare several techniques on the same dataset, while the final test score is computed outside the notebook. Our approach would report such patterns, but these false positives should not affect the developer's workflow.

In an adversarial setting, it is easy to trick our analysis by for example using meta-programming or coding patterns that exceed the capabilities of our analysis (e.g., store and load data in a file). Additional rules and environment models can strengthen our analysis but will not overcome fundamental limitations. A sound analysis for an adversarial setting may be possible but would likely be so restrictive or create so many false positives to render the approach impractical. Again, our analysis is not a safeguard against all kinds of cheating in data science competitions. It better serves as a lightweight checker that discourages bad practices of data leakage.

Dynamic analysis could improve accuracy in many cases, for example tracking the origin of individual rows in the data rather than our approximations in the related-data relationship. However, dynamic analysis would require the entire notebook to be executed for analysis (with the induced overhead), which can be costly in many machine learning tasks and may not be feasible when studying public notebooks that are often hard to reproduce [34].

6 RELATED WORK

Quality Assurance for Data Science Code. Prior work has noted that data science code is often of low quality—relying heavily on copied code and code clones [21], ignoring basic coding and style conventions [45], being poorly documented [5], containing frequent bugs in data transformations [47], and being hard to reproduce [34, 46]. In a well-known article, Sculley et al. [41] have argued that data science code is particularly prone to accumulating technical debt due to complexity and often poor engineering practices. Our work on data leakage explores another common quality issue that may lead to unreliable accuracy evaluations.

Static Analysis for Python. Despite its popularity, there are relatively few static analysis frameworks or tools written for Python, partly due to the difficulties of handling Python's dynamic features. Head et al. [12] implemented a static def-use analysis for Python to perform program slicing that helps data scientists clean, recover, and compare versions of code. *Scapel* [24] is a static analysis framework written in Python that integrates several common analyses (e.g., alias analysis). *NBLyzer* [44] is a framework specifically written for notebook code in Python, where they focus on supporting notebook actions (e.g., code changes, cell executions). Pysa [8] is a taint analysis tool that aims to identify potential security issues in Python code. Lagouvardos et al. [22] proposed a static shape analysis for Tensorflow programs, which is integrated into the *doop* framework. We chose to develop our own analysis, because it gives us the flexibility to tailor it for the purpose of leakage detection, where we need to track several custom relations.

Data Leakage Detection. Data leakage detection is a largely unexplored problem. Kaufman et al. [19] discuss how to manually perform analyses to detect certain kinds of data leakage in raw data, in particular label leakage. *Deepchecks* [6] is a library that aims to validate machine-learned models and data, which supports dynamic check for overlap leakage between train and validation/test set by dynamically inspecting datasets (e.g., whether test data contains rows that occur identically in training data). Closest to our work is a customized analysis to detect preprocessing leakage in the *NBLyzer* [44] framework. However, their data leakage analysis is just a demonstration for their framework and does not capture the full complexity of data leakage. For example, they do not establish data-model mappings but assume that all training/test locations are relevant globally (i.e., connected by the same/related models). They also do not actively distinguish validation data from test data, or track whether variables might be aliased. This results in a leakage specification that only poorly approximates the ground truth. Because their implementation is not publicly available, we did not compare our approach with theirs in our evaluation. In contrast to prior work, we detect multiple forms of data leakage and also detect leakage accurately even when datasets are transformed.

Provenance Tracking for Data Science Code. Provenance tracking in data science code has been studied extensively [30, 33]. Most relevant are approaches that track origins of data at the row level in data analytics code for various frameworks: Titian [16] for Spark, RAMP [14] for Hadoop, and Newt [25] for Hadoop and Hyracks. As discussed, we only statically approximate what could be tracked more accurately with dynamic record-level provenance tracking approaches, but these approaches need to instrument the frameworks to track provenance at runtime. Overall, this kind of tracking is only a building block in our leakage detection.

7 CONCLUSION

We provide a summary of common data leakage problems and propose a static analysis approach that could automatically detect them. We find that leakage issues are common in public notebooks and provide recommendations on process designs to prevent data leakage.

ACKNOWLEDGMENTS

Kästner and Yang's work was supported in part by NSF awards 1813598 and 2131477. Lewis and Brower-Sinning's work was funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center (DM22-0666).

REFERENCES

[1] Alexisbcook. 2021. Data leakage. https://www.kaggle.com/code/alexisbcook/data-leakage/tutorial

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *Proc. Int'l Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, 291–300.

[3] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.

[4] Andriy Burkov. 2020. *Machine learning engineering*. Vol. 1. True Positive Incorporated.

[5] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proc. Conf. Human Factors in Computing Systems (CHI)* (Honolulu, HI, USA). ACM Press, 1–12.

[6] Shir Chorev, Philip Tannor, Dan Ben Israel, Noam Bressler, Itay Gabbay, Nir Hutnik, Jonatan Liberman, Matan Perlmutter, Yurii Romanyshyn, and Lior Rokach. 2022. Deepchecks: A Library for Testing and Validating Machine Learning Models and Data. https://doi.org/10.48550/ARXIV.2203.08491

[7] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. 2015. The reusable holdout: Preserving validity in adaptive data analysis. *Science* 349, 6248 (2015), 636–638. https://doi.org/10.1126/science.aaa9375 arXiv:https://www.science.org/doi/pdf/10.1126/science.aaa9375

[8] Facebook. 2022. Facebook/pyre-check: Performant type-checking for python. https://github.com/facebook/pyre-check

[9] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. 2020. Shortcut learning in deep neural networks. *Nature Machine Intelligence* 2, 11 (2020), 665–673.

[10] Kyle Gorman and Steven Bedrick. 2019. We Need to Talk about Standard Splits. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 2786–2791. https://doi.org/10.18653/v1/P19-1267

[11] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A Survey of the State-of-the-Art. *Knowl. Based Syst.* 212 (2021), 106622.

[12] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proc. Conf. Human Factors in Computing Systems (CHI)*. 1–12.

[13] Geoff Hulten. 2018. *Building Intelligent Systems: A Guide to Machine Learning Engineering*. Apress.

[14] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 273–283. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper37.pdf

[15] imbalanced learn. 2022. common pitfalls and recommended practices. https://imbalanced-learn.org/stable/common_pitfalls.html

[16] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (nov 2015), 216–227. https://doi.org/10.14778/2850583.2850595

[17] Kaggle. 2022. House prices - advanced regression techniques. https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques

[18] Kaggle. 2022. Titanic - Machine Learning from Disasters. https://www.kaggle.com/c/titanic/code?competitionId=3136

[19] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 1–21.

[20] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Proc. Conf. Human Factors in Computing Systems (CHI)*. 1–11.

[21] Andreas P Koenzen, Neil A Ernst, and Margaret-Anne D Storey. 2020. Code duplication and reuse in Jupyter notebooks. In *Proc. Int'l Symp. Visual Languages and Human-Centric Computing (VLHCC)*. IEEE Computer Society, 1–9.

[22] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[23] Russell V Lenth. 2001. Some Practical Guidelines for Effective Sample Size Determination. *The American Statistician* 55, 3 (2001), 187–193. https://doi.org/10.1198/000313001317098149 arXiv:https://doi.org/10.1198/000313001317098149

[24] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The Python Static Analysis Framework. *arXiv preprint arXiv:2202.11840* (2022).

[25] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable Lineage Capture for Debugging DISC Analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 17, 15 pages. https://doi.org/10.1145/2523616.2523619

[26] Yingzhe Lyu, Heng Li, Mohammed Sayagh, Zhen Ming (Jack) Jiang, and Ahmed E. Hassan. 2021. An Empirical Study of the Impact of Data Splitting Decisions on the Performance of AIOps Solutions. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 54 (jul 2021), 38 pages. https://doi.org/10.1145/3447876

[27] Microsoft. 2022. Microsoft/pyright: Static type checker for python. https://github.com/microsoft/pyright

[28] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc., USA.

[29] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. https://arxiv.org/abs/2110.10234

[30] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, and Yinghui Wu. 2020. Vamsa: Tracking Provenance in Data Science Scripts. *CoRR* abs/2001.01861 (2020). arXiv:2001.01861

[31] NPR. 2022. A Tesla driver is charged in a crash involving autopilot that killed 2 people. https://www.npr.org/2022/01/18/1073857310/tesla-autopilot-crash-charges

[32] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147.

[33] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: A tool for collecting, analyzing, and managing provenance from python scripts. *Proceedings of the VLDB Endowment* 10, 12 (2017).

[34] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of Jupyter notebooks. In *Proc. Conf. Mining Software Repositories (MSR)*. IEEE Computer Society, 507–517.

[35] Cedric Renggli, Bojan Karlaš, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Continuous Integration of Machine Learning Models with ease.ml/ci: A Rigorous Yet Practical Treatment. In *Proceedings of Machine Learning and Systems*.

[36] Reuters. 2018. Amazon scraps secret AI recruiting tool that showed bias against women. https://www.reuters.com/article/us-amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G

[37] Stuart J. Russell and Peter Norvig. 1995. *Artificial Intelligence: A Modern Approach*.

[38] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 598–608. https://doi.org/10.1109/ICSE.2015.76

[39] Sohrab Saeb, Luca Lonini, Arun Jayaraman, David C. Mohr, and Konrad Paul Kording. 2017. The need to approximate the use-case in clinical machine learning. *GigaScience* 6 (2017), 1 – 9.

[40] scikit learn. 2022. common pitfalls and recommended practices. https://scikit-learn.org/stable/common_pitfalls.html

[41] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS*.

[42] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[43] Anders Søgaard, Sebastian Ebert, Jasmijn Bastings, and Katja Filippova. 2021. We Need To Talk About Random Splits. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Association for Computational Linguistics, Online, 1823–1832. https://doi.org/10.18653/v1/2021.eacl-main.156

[44] Pavle Subotic, Lazar Milikic, and Milan Stojic. 2021. A Static Analysis Framework for Data Science Notebooks. *CoRR* abs/2110.08339 (2021). arXiv:2110.08339 https://arxiv.org/abs/2110.08339

[45] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In *Proc. Int'l Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM Press, 53–56. https://doi.org/10.1145/3377816.3381724

[46] Jiawei Wang, KUO Tzu-Yang, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, 138–149.

[47] Chenyang Yang, Shurui Zhou, Jin L.C. Guo, and Christian Kästner. 2021. Subtle Bugs Everywhere: Generating Documentation for Data Wrangling Code. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA.