Statically Detecting Data Leakages in Data Science Code

Pavle Subotić
pavlesubotic@microsoft.com
Microsoft
Serbia

Uroš Bojanić
a-ubojanic@microsoft.com
Microsoft
Serbia

Milan Stojić
milan.stojic@microsoft.com
Microsoft
Serbia

Abstract
Data leakage is a well-known problem in machine learning. Data leakage occurs when information from outside the training dataset is used to create a model. This phenomenon renders a model excessively optimistic or even useless in the real world since the model tends to leverage greatly on the unfairly acquired information. To date, detection of data leakages occurs post-mortem using runtime methods. In this paper, we develop a static data leakage analysis to detect several instances of data leakages during development time. Our analysis is constructed to be lightweight so that it can be performed within interactive data science notebooks. We have integrated our analysis into the NBLyzer static analyzer framework and show its utility on real world benchmarks. To the best of our knowledge, we propose the first static detection of data science data leakages.

CCS Concepts:
• Theory of computation → Program analysis
• Software and its engineering → Integrated and visual development environments.

Keywords: static analysis, data science, data leakage

ACM Reference Format:

1 Introduction
Data science software is increasingly ubiquitous for everyday applications. Consequently, the correctness of data science software is vital considering its impact on society. While a plethora of state-of-the-art static analyzers exist that can target various programming bugs, data science programs contain domain specific bugs that are not supported by standard static analyzers. A notable data science specific bug is a data leakage [6]. Data leakages arise when external data (e.g., from the test data set) is used to train a model. Data leakages can be particularly insidious. For example, normalized input data may be split into training and testing data. However, the process of normalization transforms the data based on the entire data set (e.g., taking the average) and thus any splitting of the data cannot guarantee independence and may result in a seemingly accurate model that does not perform well in practice.

Example 1.1 (Motivating Example). Consider the program below. In the program, a CSV file is read into a data frame (line 8). From the data frame columns are selected to represent x and y coordinates (lines 14, 17). Each coordinate is normalized (lines 15) and split by selecting ranges of rows from the data frame (lines 20, 21, 23, 24). The training data is then used to train (line 30) the model and it is then tested (line 33) to evaluate its accuracy (line 36).

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB

# load dataframe
df = np.genfromtxt("data.csv", delimiter=';', dtype=None)

# preprocessing tools
min_max_scaler = MinMaxScaler()

# feature/label selection
X = df[['col1', 'col2']]
X_selected = min_max_scaler.fit_transform(X)
y = df['col3']

# train/test split
X_train = X_selected.iloc[:3]
y_train = y.iloc[:3]
X_test = X_selected.iloc[4:6]
y_test = y.iloc[4:6]

# initiate model
clf = GaussianNB()

# train model
clf.fit(X_train, y_train)

# predict labels
pred = clf.predict(X_test)

# measure score
acc = accuracy_score(y_test, pred)
print("accuracy: {}".format(acc))
```

The code above highlights the ease of inducing a data leakage. Even though the training and testing data is seemingly...
disjoint, the fact that the normalization function is called before splitting means that a data leakage is possible.

Typically, a data scientist will detect data leakage post-mortem [3]. Given a suspicious result, data analyses methods are used to identify data dependencies. These methods are powerful to explore the relationships between data, however many data leakages can be detected in the code statically and thus a static analysis can be employed as an early detection mechanism to complement postmortem data analyses techniques.

In this paper we propose a static analysis for detecting such data leakages. We present an abstract domain that tracks the origin of data frame cells to determines if two data frames originate from overlapping or tainted data sources (i.e., it has been processed by an external library function that could change the data in a way that can introduce a data leak). When a variable is an argument to a function, we assert that the variable is disjoint and untainted. For instance, in the example above, our analysis would identify that there is a potential taint between X_test and X_train since they both originate from previously normalized data, despite being disjoint.

We have implemented our analysis in the NBLyzer [12] static analysis framework for data science notebooks and evaluated its performance using 2211 real-world competition data science notebooks. The evaluation shows that our analysis performs within the time required by the use case for the vast majority of notebooks. We summarize our contributions below:

- We define a novel static analysis which detects data leakages in data science code
- We implement our analysis in the NBLyzer static analysis framework for data science notebooks
- We evaluate our analysis on real-world data science code

2 Background

2.1 Abstract State Computation

Abstract Interpretation executes the program in a soundly over-approximating semantics that ensure termination at the price of false positives. Given a sequence of statements \( s_t \), we construct a control flow graph (CFG), a directed graph that encodes the control flow of the statements. We define a CFG as \( (L, E) \) where an edge \( (l, st, l') \in E \) reflects the semantics of statement \( st \) associated with the CFG edge from locations \( l \) to \( l' \). The set of variables in all the statements is denoted by \( V \) and the set of non-variable symbols by \( S \). We assume the variables in the statements are organized in single static assignment (SSA) form [11].

A sound over-approximation \( \sigma^s \) of a state \( \sigma \) is computed by iteratively solving the semantic fixed point equation \( \sigma^s = \sigma^s_0 \cup [s_t]^s(\sigma^s) \) using the abstract semantics \([s_t]^s\) for a sequence of statements \( s_t \) and the initial abstract state \( (\sigma^s_0) \).

2.2 Data Frames

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column. Data frames have non-empty column names, with unique row names/indexes. The data stored in a data frame can be of numeric, factor or character type. Each column should contain the same number of data items. Thus semantically, a data frame has four components, namely, a data frame variable or label \( y \), a set of rows \( \bar{r} \subseteq R \), a set of columns labels \( \bar{c} \subseteq C \) and the contents of the data frame. Since we are not concerned with the contents of a data frame in this work, we ignore this component. We denote a data frame as \( y|\bar{r},\bar{c} \).

3 Data Leakage Analysis

3.1 Simplified Syntax

We define a simplified syntax for succinctly describing our analysis abstract semantics for a data frame inspired language. We define five classes of statements that are used to define the abstract semantics for our analysis.

1. source:

   \[ y = \text{read}(\text{name}) \]
   
   where \( \text{name} \in S \)

2. select-project:

   \[ y = x.\text{sel}([\bar{r}])[\bar{c}] \]
   
   where \( \bar{r} \subseteq R, \bar{c} \subseteq C \)

3. operations:

   \[ y = \text{op}(x_1, x_2) \]
   
   where \( \text{op} \in \{ \text{union, merge, diff} \} \) and \( x_1, x_2 \in V \)

4. functions:

   \[ y = f(x) \]
   
   where \( f \in \{ \text{norm, other} \} \) and \( x \subseteq V \)

5. sink:

   \[ f(x) \]
   
   where \( f \in \{ \text{test, train} \} \) and \( x \subseteq V \)

The source statement reads from a source (indicated by a label \( \text{name} \)) and stores a data frame into a variable \( y \). For example, this statement corresponds to pandas statements such as \text{read\_csv}, \text{read\_excel}, \text{read\_\_fwf} etc.

The select-project statement, creates a data frame in variable \( y \) that holds a subset of data in \( x \), based on a set of rows \( \bar{r} \) and columns \( \bar{c} \). It loosely corresponds to statements such as \text{iloc}, \text{loc} etc. commonly found in libraries like pandas and to select and project operations in relational algebra [9].

The operations class of statements transform data frames. Here we assume binary operations. Depending on the function, these have different semantics. The union statement performs a set union of two data frames. This maps to pandas statements such as \text{concat} etc. The merge statement combines two data frames by element. This maps to pandas
statements such as `join` etc. The `diff` performs a set difference of two data frames. This maps to pandas statements such as `subtract` etc.

The functions statement class models external library functions that are not atomic operations. We distinguish between two types of functions. The norm functions and other functions. By norm, we mean a function that can invoke arbitrary dependencies between cells e.g., normalization function etc. and this cannot guarantee disjointness when split.

A sink statement represents a usage of data frames (as an argument) to either train or test a model.

3.2 Abstract Domain

In this subsection we describe our abstract domain for tracking the origin of data frames in variables.

3.2.1 Abstract Data Frame.

Columns: We first describe an abstract domain to track columns in a data frame. Note, since we do not assume a schema, we do not know a priori which columns are in a given data source. We have observed that since columns labels are almost always strings, it is very rare to specify inclusion or removal as numerical indexes. For this reason we propose an abstraction that is set based. Given a set of column labels $C$, we also introduce a set of negative dual labels in a set $\bar{C}$. For a positive element $c \in C$ we denote the negative dual element as $\bar{c} \in \bar{C}$. As is the case with negation $\bar{\bar{c}} = c$, which also holds for sets i.e., $\bar{\bar{C}} = C$. In other words, the dual of the dual label is the original label. We define a abstract domain $\langle Col, \subseteq Col \rangle$ for tracking the columns in a data frames, namely,

$$Col = \{ C' | \forall c_1 \in C', \forall c_2 \in C', c_1 \cup c_2 \in C' \}$$

We say $C' \subseteq Col$ if $C' \subseteq C'$ (note the inverse relationship), that is an element with more information about included or excluded columns is more precise than a column sets with less information. The empty set is the least precise i.e., has no information on what columns may exist and a set of all columns (represented as positive or (exclusively) negatives) is the most precise. We also define an operator $Red_{Col}$ that reduces a non-canonical set to a canonical set by applying the rule $\bar{c} \cup \{ x, \bar{x} \} = \bar{c}$. We define $\cup_{Col}$ operator as $\bar{\bar{c}} \cup c \equiv c \cup \bar{\bar{c}}$ as $\cap_{Col}$ operator as $\cap$.

Example 3.1 (Column Domain). Consider the column set $C = \{ id, name, city, country, zip \}$. The abstract column set elements $C_1 = \{ id, name, city \}, C_2 = \{ id, country, city \}$. Here $C_1$ asserts that the columns are `id` and name columns. It does not have a `city` column and it may have a `country` and `zip` column. $C_2$ asserts that the columns are `id` and `country` and `city`. It may have all the other columns. $C_1 \cup_{Col} C_2 = \{ id, name, country \}$. Note that the intermediate result is reduced to not contain `city`. The join, $C_1 \cap_{Col} C_2 = \{ id \}$.

3.2.2 Data Frames: We now bring together the abstract domains for columns and rows and define an abstract domain for data frames. We define the abstract data frame as a triple:

$$(\nu, [l, u], \bar{c}) \in L = (V \cup S) \times Int_{\bar{c}} \times Col$$

For succinctness we use the notation $\nu^{\bar{c}}_{[l, u]}$ for an abstract data frame. We define join ($\sqcup_{L}$) and meet ($\sqcap_{L}$) operators as follows:

$$\nu_{[l, u]}^{\bar{c}} = \nu_{[l, u]}^{\bar{c}}$$

Note, joins and meets are performed on data frames from the same sources. We also define an overlap ($\sqcap_{L}$) predicate that asserts if two data frames from the same source overlap.

Moreover, we define a constraint function $\equiv^{\bar{c}}_{[l, u]}$ that constrains data frames from an interval and abstract column set.

We also define a minus operator on data frames, namely $\ominus$:

$$\nu_{[l, u]}^{\bar{c}} = \left\{ \begin{array}{ll}
\nu_{[l, u]}^{\bar{c}} & l_1 < l_2 \leq u_1 \leq u_2 \\
\nu_{[l, u]}^{\bar{c}} & l_2 \leq l_1 \leq u_2 \leq u_1 \\
\nu_{[l, u]}^{\bar{c}} & l_1 < l_2 \leq u_2 \leq u_1 \\
\nu_{[l, u]}^{\bar{c}} & l_2 \leq l_1 \leq u_1\end{array} \right.$$
**Data Frame Sets**: We lift the abstract data frame domain to sets that is we define \((L, \sqsubseteq_L)\) where \(L = \varphi(L)\) ordered by \(\sqsubseteq\) and define as follows \(\sqsubseteq\) and where

\[
\begin{align*}
\sqcup_L &= \text{Red}^{\sqcup_L} \circ \cup \\
\sqcap_L &= \text{Red}^{\sqcap_L} \circ \cap
\end{align*}
\]

We define a reduction operator \(\text{Red}^{op}_L\) as:

\[
\text{Red}^{op}_L = \left\{ \begin{array}{ll}
\{ l_1 \mid l_1, l_2 \in L.\text{lok}(l_1, l_2) \} \cup \\
\{ l_1 \mid l_1 \in L.\neg\exists l_2 \in L.\text{lok}(l_1, l_2) \}
\end{array} \right.
\]

Essentially, after a join or meet is performed (\(op\)) the set of data frames may overlap and thus the set needs to be put into a canonical form. We also lift the \(\sqcup\) and \(\sqcap\) operators to sets where the set version perform the operations for each element in the set. Since \(\sqcap\) returns a set, a flattening is performed.

**Example 3.3** (Data Frame Sets). Consider the data frame sets \(\{\text{file}_{[1,10]}^{\text{id}}\} \sqcup \{\text{file}_{[0,100]}^{\text{name}}\}\) and \(\{\text{file}_{[1,12]}^{\text{id}}, \text{file}_{[0,100]}^{\text{name}}, \text{file}_{[0,100]}^{\text{zip}}\}\). If we naively perform a join, we can get elements which can be joined into a single data frame i.e.,

\[
\{\text{file}_{[1,10]}^{\text{id}}, \text{file}_{[1,12]}^{\text{id}}, \text{file}_{[0,100]}^{\text{name}}, \text{file}_{[0,100]}^{\text{zip}}\}
\]

The reduction makes the set canonical i.e.,

\[
\{\text{file}_{[1,12]}^{\text{id}}, \text{file}_{[0,100]}^{\text{name}}, \text{file}_{[0,100]}^{\text{zip}}\}
\]

Note, if we do not perform this reduction, it potentially hinders performance and violation detection.

### 3.2.2 Abstract State

We define the abstract state as a mapping \(V \rightarrow D\) between variables in \(V\) and an abstract domain where

\[
D = (\bar{L} \times \bar{B} \times \varphi(\{\text{tr.ts}\})), \sqsubseteq
\]

where we define \(\sqsubseteq = \sqsubseteq_L \times \sqsubseteq_{\bar{B}} \times \sqsubseteq\) and \(\cup = \cup_L \times \cup_{\bar{B}} \times \cup\) and \(\cap = \cap_L \times \cap_{\bar{B}} \times \cap\).

Intuitively, for variable \(v\) we keep track of (1) the set of data frames it is dependent on, (2) if those data frames are tainted i.e., have data that cannot be safely decomposed and (3) if any data frames have been used for training or testing. We differentiate between which tuple element in the product domain is access by \(\sigma^{\#1}\) (first tuple element) and \(\sigma^{\#2}\) (second tuple element) and \(\sigma^{\#3}\) (third tuple element).

Since data frames can have infinite ascending chains w.r.t the row interval abstraction and column abstraction, a simple widening operator can be applied for rows [5] and similarly, for columns the set can be set to \(\emptyset\) (the top element) if the set of columns are detected to be reducing inside a loop.

### 3.2.3 Knowledge Base

We also assume knowledge bases \(KB_{\text{source}}, KB_{\text{norm}}\) and \(KB_{\text{test}}, KB_{\text{train}}\) which holds functions that act as a source, introduce data leaks, and be used for testing and training, respectively.

### 3.3 Abstract Semantics

We define an abstract semantics for our data frame language. The abstract semantics for each category of statements is described below.

1. **source**:

\[
\lambda \sigma^y \cdot [y = \text{read}(\text{name})] = \sigma^y[y \mapsto (\{\text{name}^{\emptyset}_{[0,\infty]}\}, F, 0)]
\]

where \(\text{read} \in KB_{\text{source}}\)

2. **select-project**:

\[
\lambda \sigma^y \cdot [y = x.\text{sel}(r)[c]] = \\
\lambda \sigma^y[y \mapsto (\sigma^{\#1}(x) \cup \lambda \sigma[y \mapsto (\sigma^{\#2}(x), \sigma^{\#3}(x))])
\]

3. **operations**:

\[
\lambda \sigma^y[y = \text{op}(x_1, x_2)] = \\
\text{a. op = union:} \\
\sigma^y[y \mapsto \sigma^y(x_1) \cup \sigma^y(x_2)] \\
\text{b. op = merge:} \\
\left\{ \begin{array}{l}
\sigma^y[y \mapsto \sigma^y(y) \cup \\
\sigma^y(x_1) \cup \beta \sigma^y(x_2) \\
\sigma^y(x_1) \cup \beta \sigma^y(x_2) \\
\sigma^y(x_1) \cup \beta \sigma^y(x_2)
\end{array} \right.
\]

\text{c. op = diff:} \\
\left\{ \begin{array}{l}
\sigma^y[y \mapsto \sigma^y(y) \cup \\
\sigma^y(x_1) \cup \beta \sigma^y(x_2) \\
\sigma^y(x_1) \cup \beta \sigma^y(x_2) \\
\sigma^y(x_1) \cup \beta \sigma^y(x_2)
\end{array} \right.
\]

4. **functions**:

\[
\lambda \sigma^y \cdot [y = f(x)] = \\
\text{a. } f \in KB_{\text{train}}: \\
\forall x \in x.\sigma^y[y \mapsto (\sigma^{\#1}(y) \cup T, \sigma^{\#3}(x))] \\
\text{b. otherwise:} \\
\sigma^y[y \mapsto \sigma^y(y) \cup \left\{ \begin{array}{l}
\sigma^y(x)\end{array} \right\}
\]

5. **sink**:

\[
\lambda \sigma^y \cdot [f(x)] = \\
\forall x \in x.\sigma^y[x \mapsto (\sigma^{\#1}(x), \sigma^{\#2}(x) \cup \{tr\})] \\
\text{if } f \in KB_{\text{train}} \\
\forall x \in x.\sigma^y[x \mapsto (\sigma^{\#1}(x), \sigma^{\#2}(x) \cup \{ts\})] \\
\text{if } f \in KB_{\text{test}}
\]
In the above abstract semantics the read statement produces a mapping between the data frame variable \( y \) and the file name. Because we do not statically know any information about the file data, we simply do not constrain columns (empty set) or rows (unbounded interval).

For the union, merge and diff statement, we perform a corresponding data frame operation on the two data frames \( x_1 \) and \( x_2 \), namely, join, meet and minus and add it to the previous state of \( y \).

For the norm function, we reset the mapping, by setting the outgoing variable as the source mapping using the rename (\( rn \)) function which simply renames all source variables in the first argument to the variable in the second argument. We omit the definition of \( rn \) as it is intuitive. We also set the flag to true (\( T \)). If the function is not a norm function, we simply propagate the state information.

Finally in the case of a sink statement we mark each argument with a \( [tr] \) element if it is an argument to a train function, and \([ts]\) if it is an argument to a test function.

**Example 3.4 (Motivating Example (Cont.)).** Consider our motivating example again. In the program a CSV file is read into a data frame (line 8). This creates a mapping of

\[
df \mapsto ((data^0_{[0, \infty]}), F, \emptyset)
\]

From the data frame columns are selected to represent \( x \) and \( y \) co-ordinates (lines 14, 17). This adds constraints so that

\[
x \mapsto ((data^{col1, col2}_{[0, \infty]}), F, \emptyset)
\]

\[
y \mapsto ((data^{col3}_{[0, \infty]}), F, \emptyset)
\]

When each coordinate is normalized (lines 15) we flip the second tuple in the domain to \( T \) and reset the source by setting the source to the variable that holds the normalized data. When we split by selecting ranges of columns from the data frame (lines 20, 21, 23, 24) we perform a selection and get mappings

\[
X\_train \mapsto ((x\_selected^{col1, col2}_{[0,2]}), T, \emptyset)
\]

\[
X\_test \mapsto ((x\_selected^{col1, col2}_{[4,5]}), T, \emptyset)
\]

The training data is then used to train (line 30) the model and it is then tested (line 33) to evaluate its accuracy (line 36). At the end we obtain:

\[
X\_train \mapsto ((x\_selected^{col1, col2}_{[0,2]}), T, \{tr\})
\]

\[
X\_test \mapsto ((x\_selected^{col1, col2}_{[4,5]}), T, \{ts\})
\]

which violates the error conditions because they have the same source, and despite being disjoint, both passed through a function which tainted them. Consider if normalization was called after splitting. \( X\_train \) and \( X\_test \) would have different sources and be disjoint. We would then assert that no data leak occurred. Also consider there is no tainted normalizing function and no splitting. Without the flag, we would have to assert false positives (any two variables with same source can’t be trained and tested).

## 4 Integration into NBLyzer

We have integrated our analysis into NBLYZER, a static analysis framework for data science notebooks as described in [12]. The analysis technique is depicted in Figure 1.

The analyzer ingests the statements as a CFG as defined in Subsection 2.1. The analysis operates by performing standard static analysis intra cell. At the cell level this computation is defined as \( F_{c_i} \), which we refer to a abstract cell transformer. \( F_{c_i} \) takes an abstract state and computes a fix-point solution [10, 13] in the abstract domain. At each execution, a cell transformer \( F_{c_i} \) for a cell \( c_i \) is applied with the current state, returning an updated new state i.e., \( F_{c_i}(\sigma^*) = \sigma^{#'} \).

For inter cell analysis, the abstract state needs to be propagated from one cell to another for a fixed depth of \( K \) or until a notebook wide fixed point is reached (\( K = \infty \)). It relies on an inter cell dependency graph which is defined by a predicate \( \phi \). Each analysis needs to define \( \phi \) along with its abstract semantics. Moreover, each cell has preconditions \( pre_{c_i} \), typically the set of unbound variables. If \( \phi \) holds, the abstract state is propagated to the dependent cells, for which the incoming abstract state is treated as a initial state. For each cell the abstract state is checked for correctness criteria, if an error is found a report is updated which serves as instruction for notebook clients to alert the user to the consequences of the event (e.g., by cell highlighting etc.).

To integrate into the NBLYZER framework we specify the additional \( \phi \) condition for inter-cell propagation as follows:

\[
\phi(\sigma^*_{c_i}, pre_{c_i}) = pre_{c_i} \subseteq \{ \upsilon : (\upsilon \mapsto x(\upsilon, u)) \in \sigma^*_{c_i} \land [l, u] \neq \bot \} \land pre_{c_i} \neq \emptyset
\]

This rule stipulates the condition by which a successor cell should be analyzed. That is, it states that if variables with rows (do not map to \( \bot \) interval) in the abstract state of the current notebook cell, are also unbound in the successor notebook cell (\( pre_{c_i} \)), then we proceed to propagate the abstract state to that successor cell.
5 Experimental Evaluation

**Experimental Setup:** All experiments were performed in an Intel(R) Xeon W-2265 CPU @ 3.50 GHz with 64 GB RAM running a 64 bit Windows 10 operating system. Python 3.8.8 was used to execute NBLyzer. We evaluate the execution-time of our static analyses running within NBLyzer on the full set of Kaggle notebooks.

We use a benchmark suite consisting for 2211 executable real-world notebooks from the Kaggle competition[1] that has previously been used to evaluate data science static analyzers [8]. The benchmark characteristics are summarized in Table 1. We emphasise that this is a fair reflection of typical notebook code.

We see on average, the notebooks in the benchmark suite have 24 cells, where each cell on average has 9 lines of code. In addition, on average branching instructions appear in 33% of cells. Each notebook has on average 3 functions and 0.1 classes defined. We note that these characteristics of low amount of branching, functions and classes are typically advantageous for static analysis precision. We found that every second notebook had a cell that could not be parsed and analyzed due to syntax errors in it. Overall, this affected 4% of cells in the benchmarks. 2.1 of variables were unbound, from an average of 8.2 variables per cell.

**Performance Evaluation:** We evaluate the performance of the data leakage analysis. This analysis is run on the NBLyzer setting $K = \infty$ (propagate till fixed point). In Figure 2 the average and maximum data leakage analysis for executions are shown. The results show that the average data leakage analysis on a notebook takes 41.45 milliseconds. The average maximum analysis per notebook take 880.9 milliseconds, with a global maximum of 233 seconds. Since this analysis is run as a what-if analysis in a notebook, we require fast response times (ideally < 1000ms) to retain the interactive notebook experience [2]. The analysis time for average case is well under the threshold for users to notice any delay and does not degrade the user experience. The average maximum recorded analysis time is above the immediate fell threshold, but below the threshold for the task feeling out of flow (1000ms). The global maximum does cause considerable delay and user degradation. Moreover, only 4% of all analyses execute for more than 1000ms and only 1% for more than 5000ms.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Mean</th>
<th>SD</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cells (per-notebook)</td>
<td>23.58</td>
<td>20.21</td>
<td>182</td>
<td>1</td>
</tr>
<tr>
<td>Lines of code (per-cell)</td>
<td>9.12</td>
<td>13.35</td>
<td>257</td>
<td>1</td>
</tr>
<tr>
<td>Branching instructions (per-cell)</td>
<td>0.43</td>
<td>2.49</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>Functions (per-notebook)</td>
<td>3.33</td>
<td>7.11</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>Classes (per-notebook)</td>
<td>0.34</td>
<td>0.64</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Non-parsing cells (per-notebook)</td>
<td>0.3</td>
<td>0.98</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Variables (per-cell)</td>
<td>8.2</td>
<td>2.3</td>
<td>552</td>
<td>0</td>
</tr>
<tr>
<td>Unbound variables (per-cell)</td>
<td>2.1</td>
<td>1.06</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. Kaggle Notebook Benchmark Characteristics

6 Related Work

**Static Analysis for Data Science.** Static analysis for data science is an emerging area in the program analysis communities. A comprehensive state of the art is outlined in [14]. In [12] is a static analysis framework for data science notebooks. We implement our technique inside this framework. Our analysis can benefit from analyses such as the one presented in [7] and [15].

**Data Leakage Detection and Avoidance.** Several techniques exist to avoid and discover data leakages in data science code. For example, a popular technique is the use of data science pipelines [4] that stage the phases of sourcing, cleaning, splitting, normalization, and training to avoid performing a normalization step before splitting. This, however, requires manual effort and is not widely used among the millions of data scientists. For this reason tools such as [3] exist that perform dynamic instrumentation to detect a data leakage.

7 Conclusion

We have presented a method for detecting data science data leakages statically. Our technique has been integrated into the NBLyzer static analyzer. We believe this is the first static analysis for detecting data science data leakages. In our current prototype we have not implemented all operations but have reasonable coverage of pandas data frames. Our analysis does not model several dynamic language constructs. However, we believe in principle given an appropriate (light-weight) alias analysis this can be achieved. We also note that data science code is simple compared to mainstream programs, and rarely employs dynamic features.

**Acknowledgments**

We thank our colleagues at Microsoft Azure Data Labs and Gray Systems Lab for all their feedback and support.
References


