LeakageDetector: An Open Source Data Leakage Analysis Tool in Machine Learning Pipelines

Eman Abdullah AlOmar*, Catherine DeMario*, Roger Shagawat*, Brandon Kreiser*
*Stevens Institute of Technology, Hoboken, New Jersey, USA
{ealomar,cdemario,rshagawa,bkreiser}@stevens.edu

***Abstract*—Code quality is of paramount importance in all types of software development settings. Our work seeks to enable Machine Learning (ML) engineers to write better code by helping them find and fix instances of Data Leakage in their models. Data Leakage often results from bad practices in writing ML code. As a result, the model effectively "memorizes" the data on which it trains, leading to an overly optimistic estimate of the model performance and an inability to make generalized predictions. ML developers must carefully separate their data into training, evaluation, and test sets to avoid introducing Data Leakage into their code. Training data should be used to train the model, evaluation data should be used to repeatedly confirm a model's accuracy, and test data should be used only once to determine the accuracy of a production-ready model. In this paper, we develop LEAKAGEDETECTOR, a Python plugin for the PyCharm IDE that identifies instances of Data Leakage in ML code and provides suggestions on how to remove the leakage. The plugin and its source code are publicly available on GitHub at https://github.com/SE4AIResearch/DataLeakage_Fall2023. The demonstration video can be found on YouTube: https://youtu.be/yXj3wihSaIU.**

***Index Terms*—data leakage, machine learning, quality**

I. INTRODUCTION

Data Leakage is one of the critical issues in writing Machine Learning (ML) that can significantly affect the performance and reliability of models [5], [21], [10], [2], [7], [14], [8], [11], [13], [17], [3]. It occurs when information outside the training dataset is inadvertently used to create the model, leading to overly optimistic performance and poor generalization to new data. Data Leakage often arises from bad practices in data handling and code implementation, making it a pervasive challenge in the field [16], [9], [1], [6], [10], [18]. Previous studies have shown that Data Leakage is a prevalent bad practice, particularly in educational settings [15], [5].

Recently, Yang *et al.* [21] have proposed a static analysis approach to detect Data Leakage, that is a command-line tool that can detect common forms of Data Leakage (*i.e.,* Overlap, Multi-test, and Preprocessing) in data science code. However, despite efforts to engineer an automated detection for Data Leakage, the tool is designed for research purposes, such as enabling large-scale empirical studies. The tool could benefit from some aspects, such as usability, user engagement, and leakage correction, which enhances practitioners' adoption.

To cope with these challenges, this paper aims to support developers with the identification of Data Leakage by designing LEAKAGEDETECTOR, a PyCharm IDE plugin that detects and proposes quick fixes for Data Leakage instances.

Our plugin enhances the functionality of the leakage static analysis tool [21] by integrating directly with the PyCharm IDE, making it more user-friendly and accessible. Unlike the original tool, which requires command-line proficiency and manual HTML file review, our plugin operates within the IDE with a single click, providing clear, plain-language descriptions of different types of leakage. It simplifies deployment, scales well with the user base due to PyCharm's popularity, and offers customization features such as quick fixes and "TODO" notes.

As an illustrative example, Figure 1, shows a step-by-step scenario to automatically detect and fix the Data Leakage instance. Once the user opens a Python file within PyCharm, and opens the Data Leakage Analysis Tool Window, the user clicks on the "Run Data Leakage Analysis" button to run the leakage analysis tool ❶. When leakage analysis is complete ❷ and code segments associated with Data Leakage are highlighted, the same code segments are shown in the Tool Window ❸. The user selects the Data Leakage instance ❹ and can hover over the highlighted code associated ❺ and apply the quick fix ❻.

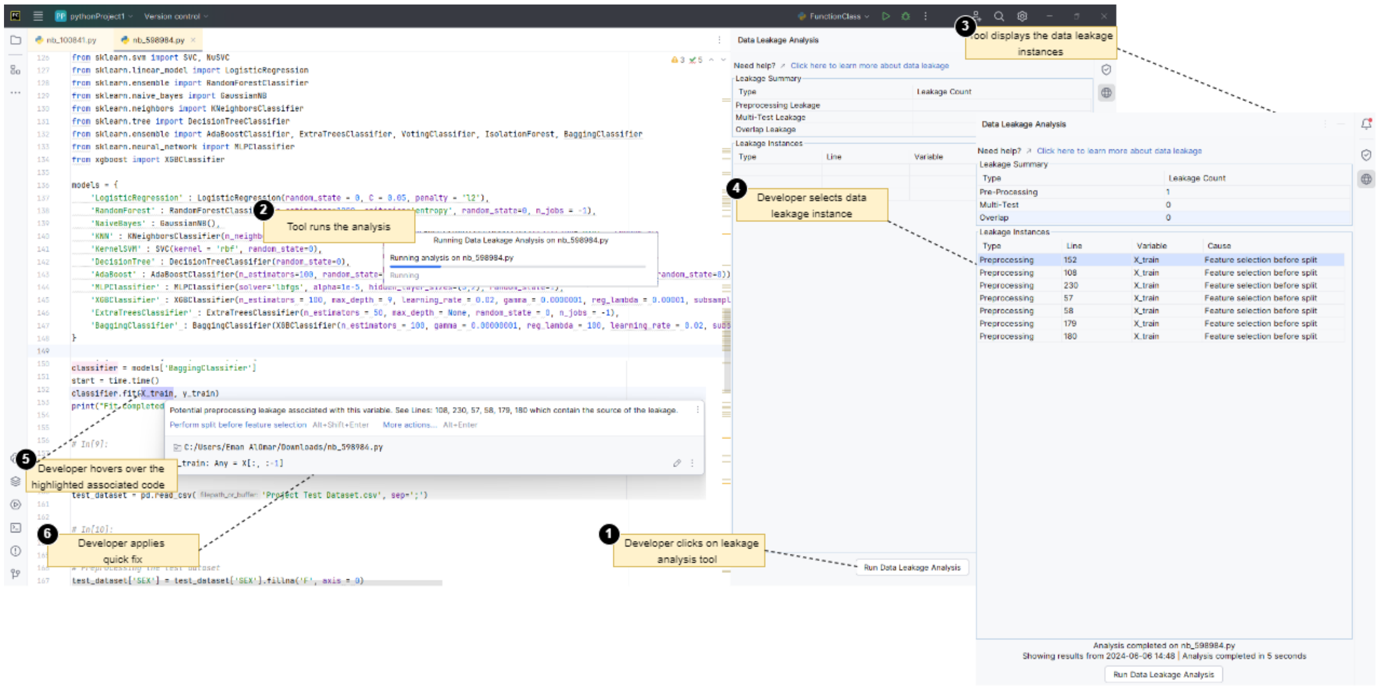
The preliminary evaluation of our tool shows its ability to detect and suggest quick fixes for the selected Data Leakage instances. The survey responses are promising, as the majority of participants are satisfied with the recommendations of LEAKAGEDETECTOR.

The remainder of this paper is organized as follows: Section II outlines our tool design. Section III discusses our preliminary evaluation, while the research discussion is addressed in Section IV. Section V reviews the existing studies related to the Data Leakage, before concluding with Section VI.

II. STUDY DESIGN

A. LEAKAGEDETECTOR *Architecture*

LEAKAGEDETECTOR is implemented as an open-source Python plugin for PyCharm IDE. A high-level overview of the architecture of LEAKAGEDETECTOR is depicted in Figure 2. While the user works on the PyCharm IDE with the plugin installed, the user can navigate to the Data Leakage Tool Window on the screen's right side. Located at the bottom of the Tool Window, there is a button labeled "Run Data Leakage Analysis", which invokes the following process: The plugin checks whether the Docker image of the Leakage Analysis tool [21] is downloaded on the user's machine. Once the image is on the user's machine, the plugin connects to the


pythonProject1
Version control
FunctionClass
nb_100841.py
nb_598984.py
1 Developer clicks on leakage analysis tool
2 Tool runs the analysis
3 Tool displays the data leakage instances
4 Developer selects data leakage instance
5 Developer hovers over the highlighted associated code
6 Developer applies quick fix
Run Data Leakage Analysis

```
from sklearn.svm import SVC, NuSVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier, ExtraTreesClassifier, VotingClassifier, IsolationForest, BaggingClassifier
from sklearn.neural_network import MLPClassifier
from xgboost import XGBClassifier

models = {
    'LogisticRegression' : LogisticRegression(random_state = 0, C = 0.05, penalty = 'l2'),
    'RandomForest' : RandomForestClassi                        entropy', random_state=0, n_jobs = -1),
    'NaiveBayes' : GaussianNB(),
    'KNN' : KNeighborsClassifier(n_neighb
    'KernelSVM' : SVC(kernel = 'rbf', random_state=0),
    'DecisionTree' : DecisionTreeClassifier(random_state=0),
    'AdaBoost' : AdaBoostClassifier(n_estimators=100, random_state=                                  random_state=0))
    'MLPClassifier' : MLPClassifier(solver='lbfgs', alpha=1e-5, hid
    'XGBClassifier' : XGBClassifier(n_estimators = 100, max_depth = 9, learning_rate = 0.02, gamma = 0.0000001, reg_lambda = 0.00001, subsampl
    'ExtraTreesClassifier' : ExtraTreesClassifier(n_estimators = 50, max_depth = None, random_state = 0, n_jobs = -1),
    'BaggingClassifier' : BaggingClassifier(XGBClassifier(n_estimators = 100, gamma = 0.00000001, reg_lambda = 100, learning_rate = 0.02, subs
}

classifier = models['BaggingClassifier']
start = time.time()
classifier.fit(X_train, y_train)
print("Fit completed

# In[9]:

test_dataset = pd.read_csv( filepath_or_buffer: 'Project Test Dataset.csv', sep=';')


# In[10]:

# Preprocessing the test dataset
test_dataset['SEX'] = test_dataset['SEX'].fillna('F', axis = 0)
```

```
Running Data Leakage Analysis on nb_598984.py
Running analysis on nb_598984.py
Running
```

```
Potential preprocessing leakage associated with this variable. See Lines: 108, 230, 57, 58, 179, 180 which contain the source of the leakage.
Perform split before feature selection  Alt+Shift+Enter    More actions...  Alt+Enter
C:/Users/Eman AlOmar/Downloads/nb_598984.py
_train: Any = X[:, :-1]
```

Data Leakage Analysis

Need help? Click here to learn more about data leakage

Leakage Summary

Type	Leakage Count
Pre-Processing	1
Multi-Test	0
Overlap	0

Leakage Instances

Type	Line	Variable	Cause
Preprocessing	152	X_train	Feature selection before split
Preprocessing	108	X_train	Feature selection before split
Preprocessing	230	X_train	Feature selection before split
Preprocessing	57	X_train	Feature selection before split
Preprocessing	58	X_train	Feature selection before split
Preprocessing	179	X_train	Feature selection before split
Preprocessing	180	X_train	Feature selection before split

Analysis completed on nb_598984.py
Showing results from 2024-06-06 14:48 | Analysis completed in 5 seconds

Figure 1: LEAKAGEDETECTOR in action, showing the identified Data Leakage instances.

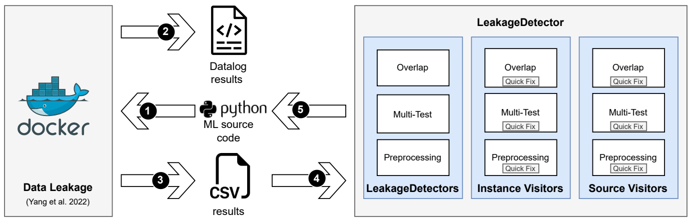


Figure 2: High-level architecture of LEAKAGEDETECTOR.

Docker engine, creates a container from the image, and uses that container to analyze the currently open Python file. Then, the output from the tool is placed in a temporary folder on the user's machine. Our plugin classes contain functionality that examines these files to determine whether leakage is present in the user's Python file. The plugin's inspection visitors will use the information from the leakage detectors to determine where to render inspections in the Python file. If there is a leakage, the plugin will perform inspections (*i.e.,* highlight/underline) of the code involved in the leakage. The user can apply a quick fix to a piece of code involved in the leakage. The plugin keeps track of the lines affected by the quick fix in a file on the user's machine.

Our plugin builds on the functionality of the leakage static analysis tool [21] in a number of key ways:

- **Functionality.** Compared to the Data Leakage static analysis tool, our plugin can be run within the IDE. Originally, if a user wanted to run the tool, they would have to run a command line and then open an HTML file with the output. Having the tool already in the IDE makes the tool much easier for anyone to use.
- **User Interaction.** The Data Leakage static analysis tool requires users to be proficient in using command-line commands and SWIG to run the tool. This can be a problem for those who are new to coding. Once installed in PyCharm and alongside a running Docker, our plugin requires just a push of a button to run. Our plugin easily displays plain-language descriptions of the different types of leakage in the IDE for users to see and understand.
- **Integration.** The plugin is already integrated with PyCharm compared to the original tool, which has no integration with any IDE.
- **Ease of Deployment.** The plugin does require some extra steps, such as installing Docker and installing the plugin in PyCharm; compared to the tool, it is just a couple of command lines. Yet, once all the prerequisites to the plugin are installed, the plugin is easier to use, as the plugin can run with a click of a button.
- **Scalability.** Both the plugin and the tool have a similar scale in terms of the amount of code they can detect leakage on (*i.e.,* 1 file). Regarding audience reach, we believe that the plugin would be more accessible to a greater audience as PyCharm is a popular IDE machine-learning tool that developers like to use.
- **Customization.** The plugin and the tool both have the ability to identify leakages and enable users to navigate to the specific line where the error occurs. the plugin has the added benefit of allowing users to do a quick fix (see Subsection II-B), which would change the code and add a "TODO" for the users explain what needs to be fixed to improve their code.
- **Feedback Mechanism.** Both the plugin and the tool show

similar results (*e.g.,* type of Data Leakage, line number) in a table. Still, the tool shows the table at the top of the HTML result, and when a user scrolls down, the result table does not follow. On the other hand, the plugin would have the table displayed in the tool itself, which would be on the right side of the IDE. Also, the plugin will always be on the right side if the user scrolls down through their code.

We believe that our plugin complements the command-line-based tool [21], providing users with the flexibility to choose the option that best suits their needs and preferences for detecting leakage.

B. Quick Fixes

1) Overlap Leakage: It occurs when the training and test sets overlap; in other words, the two sets share data. Quick fixes for Overlap Leakage are associated with the sources of Overlap Leakage. Quick fixes may be applied to Overlap Leakage sources whose causes are *SplitBeforeSample*, *i.e.,* splitting is performed before sampling. The plugin moves the existing call to a function with the keyword "split" above the nearest existing call to the function with the keyword "sample". The plugin also adds a "TODO" message above the new call to "split" that instructs the user to check their code. The applied fix may not be syntactically or semantically correct, as the user must complete the fix. An example of an instance of Overlap Leakage (after the quick fix has been applied) is shown below in Listing 1.

```
#TODO: Check the arguments provided to the
     call to split.
X_train, X_test, y_train, y_test =
    train_test_split(
X_new, y_new, test_size =0.2, random_state
     = 42)
X_new, y_new =
     SMOTE().fit_resample (X,y)
```

Listing 1: A quick fix applied to Overlap Leakage.

2) Multi-test Leakage: It occurs when the same test data is used in multiple evaluations. To apply a quick fix for Multi-test Leakage, the user hovers over one of the variables associated with a particular instance of Multi-test Leakage and selects "*Use new test data for each evaluation*". Each variable associated with the Multi-test Leakage will have a suffix of the form "_#" (*e.g.,* _4) where the "#" is one fewer than how many times that variable is reused. For example, if the last usage of the variable associated with an instance of Multi-test Leakage is renamed from `X_test` to `X_test_4`, `X_test` was used five times before the user applied the quick fix. The plugin also adds a "TODO" message above the newly renamed variable that instructs the user to load their test data into that variable. An example of a Multi-test Leakage instance (after the quick fix has been applied) is shown in Listing 2.

```
#TODO: Load the test data for the
    evaluation.
lr_score = lr.score (X_test_0 , y_test)
```

Listing 2: A quick fix applied to Multi-test Leakage.

3) Preprocessing Leakage: It occurs when training and test data are transformed together. Quick fixes for Preprocessing Leakage are associated with the sources of Preprocessing Leakage. The plugin handles two distinct scenarios where Preprocessing Leakage occurs: (1) a split is performed after feature selection, and (2) no split is performed. To address the first case, the quick fix *moves the split before the feature selection*. To address the second case, *a new line of code must be added to split the data*. In essence, if a split call is already present, the quick fix simply moves that split call above the feature selection. If there is no split call present, the plugin adds a line containing SPLIT() above the feature selection. The plugin also adds a "TODO" message above the new call to "split" that instructs the user to check their code. It is up to the user to complete the code and ensure that it is correct. An example of a preprocessing leakage (after a quick fix has been applied) is shown in Listing 3.

```
#TODO: Check the arguments provided to the
     call to split.
split()
wordsVectorizer = CountVectorizer() .fit(
    journalsFinal['text'])
wordsVector = wordsVectorizer.transform(
    journalsFinal['text'])
```

Listing 3: A quick fix applied to Preprocessing Leakage.

It is worth noting that quick fixes simply provide a skeleton for a true fix. They are not substitutes for human judgement. The user needs to complete the fix and ensure the syntactical and semantical correctness of their code. A comprehensive fully automated solution is challenging, as it requires a deep conceptual understanding of the problem at hand. For example, when selecting features to use for model training, a developer might erroneously choose to use all features, rather than only a subset of features. Consider, for example, a model that is intended to predict the yearly salary of a person based on a variety of factors. If a data point such as the monthly salary of the person appears in the training dataset, the model will simply use the monthly salary as an indicator of the yearly salary and ignore the other factors [5]. A model trained on the wrong features, or more features than necessary, will not perform well in a production environment.

III. Preliminary Evaluation

We conducted a study to evaluate the effectiveness of LeakageDetector in detecting Data Leakage and fixing them. Since LeakageDetector is built on top of the leakage static analysis tool, we did not evaluate the detection accuracy of the tool independently. Instead, we considered the performance reported by Yang *et al.* [21] to be valid for our purposes as well.

Since there is only one existing publicly available dataset [21] that contains Data Leakage, we decided to use it for our own validation set. We started by selecting 31 Python files and manually analyzing the existing annotated sets and running them using our plugin. We invited 8 participants from Stevens Institute of Technology to use any of these files when trying our plugin. All participants volunteered for the experiment, and informed consent was obtained. The industrial experience of the respondents ranged from 1 to 6 years, their ML experience ranged from 1 to 6 years, their expertise in programming ranged from 1 to 6 years, and their expertise in Python ranged from 1 to 6 years. Before the experiment, the participants received an hour-long tutorial on Data Leakage along with reference materials.

To evaluate the usefulness of our plugin, we have posted a survey for users to take it optionally. The survey consisted of 15 questions that are divided into 2 parts. The first part of the survey includes demographic questions about the participants. In the second part, we ask about the (1) found Data Leakage type, (2) level of satisfaction with the tool, (3) additional information to be added to the tool window, (4) feasibility of implementing the quick fixes, and (5) overall impression of the tool and the proposed idea for improvement. As suggested by Kitchenham and Pfleeger [12], we constructed the survey to use a 5-point ordered response scale ("Likert scale") question on the level of satisfaction with the tool, 5 open-ended questions on the tool experience and challenges, and 2 multiple choice questions on the type of Data Leakage found and the ability to see the tool window with all related information. All participants completed the survey.

We report the results of the users who have taken it in Figures 3 and 4. Overall, as illustrated in Figure 3, the proportions of encountering different types of Data Leakage were similar for Overlap and Multi-test leakages. However, the most frequently observed was Preprocessing leakages, each accounting for 55.6% of the total cases. This is expected as this type of leakage is more readily made in code than Multi-test. Multi-test leakage deals with one variable being used multiple times. A quick fix for this would be to make different variable names. Since the quick fix for the leakage is simply a name change, users who run into this problem probably just had simple naming mistakes and the fact that they need more variables. Regarding user satisfaction, the survey results varied from 'Very Satisfied', mainly with the tool setup, ease of use, execution time, format of input/output, quick fixes and highlights, and mostly neutral opinions with the tool documentation. However, a user is less satisfied with the quick fixes. The somewhat unsatisfaction can be explained by quick fixes that do not completely fix the code. A comprehensive and fully automated correction is challenging, as it requires a deep conceptual understanding of the problem.

IV. Implications

Enhancing maintenance and evolution with LeakageDetector. Our tool helps data scientists and ML model developers identify and fix instances of Data Leakage in their code. Data Leakage is common in various types of notebooks, including those used for educational purposes [21]. Data

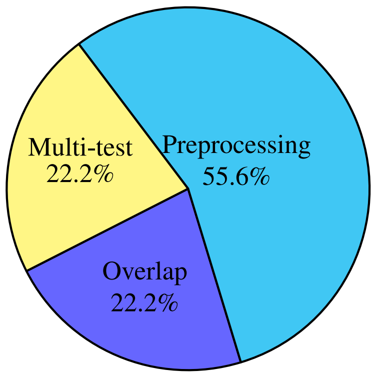


Figure 3: Distribution of Data Leakage types selected by participants.

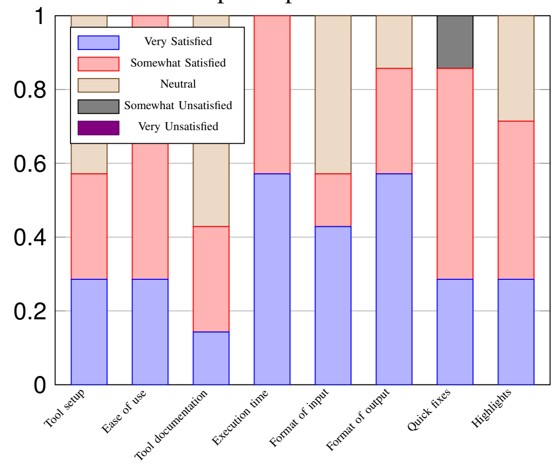


Figure 4: Participants' satisfaction with various aspects of the LeakageDetector tool.

Leakage is often the result of human oversight, making it challenging for developers to spot in their own code. Regardless of the purpose of an ML model, Data Leakage can have significant consequences. In an educational setting, learners might adopt bad habits if they see examples of code with Data Leakage. In production environments, the impact can be more immediate, potentially causing a model to underperform in critical applications.

Promoting adoption of coding best practices. Our tool is designed to facilitate extensions and integration with other tools. This makes it a valuable resource for bridging the gap between research and practice in software maintenance and evolution. By simplifying deployment and use, LeakageDetector can help practitioners access advanced tools more easily, promoting the adoption of best practices in the industry. The architecture of LeakageDetector is designed to reduce the effort required for installation and usage, making cutting-edge research tools more accessible to practitioners.

V. Related Work

Kery *et al.* [11] interviewed 21 data scientists to study coding behaviors, and how they keep track of variants they explored. Their findings stimulate the development of novel designs for interacting with notebook cells, facilitating tasks such as browsing history, debugging, and more, which could enhance the efficiency of literate programming in supporting data science endeavours. Yang *et al.* [21] proposed a static analysis approach to detect common forms of Data Leakage (*i.e.,* Overlap, Multi-test, and Preprocessing) in data science code. Bouke and Abdullah [4] investigated the impact of Data Leakage during data preprocessing on the reliability of machine learning based intrusion detection systems. The study employs six models on the datasets with and without Data Leakage to compare their performance. Apicella *et al.* [2] discussed Data Leakage issues in the machine learning pipeline. The authors classified Data Leakage in machine learning and examined how specific circumstances can spread throughout the machine learning process workflow. Drobnjaković *et al.* [7] developed a static analysis based on abstract interpretation to verify the absence of Data Leakage. Their approach was incorporated into the NBLyzer framework and was evaluated for its effectiveness and accuracy using 2111 Jupyter notebooks sourced from the Kaggle competition platform. Venkatesh *et al.* [19] introduced HeaderGen, a tool that autonomously labels code cells with categorical markdown headers using a taxonomy for machine learning operations and classifies and displays function calls based on this classification. Wang *et al.* [20] proposed Themisto toot to help write documentation for code cells by applying a deep learning-based approach to generate documentation in natural language and then recommending to the user whether to adopt it or use it directly.

VI. Conclusion and Future Work

We develop LeakageDetector, a PyCharm plugin that supports the detection and correction of Data Leakage. We conducted a preliminary assessment of the capabilities of the tool and plan to perform more empirical experiments as part of our future work. Additionally, it may be worth exploring the possibility of supporting other forms of Data Leakage. Further, the leakage analysis tool examines a variety of function calls from popular ML packages (*e.g.,* Keras, Sklearn). Our plugin provides plain language descriptions of leakage instances, identifies their sources and causes, and proposes fixes. In the future, we plan to support additional Data Leakage causes as the plain language description of a leakage instance (and its cause) depends on the function calls involved in that leakage instance.

VII. Acknowledgments

We would like to thank the authors of the leakage static analysis tool [21] for publicly providing it.

References

[1] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[2] A. Apicella, F. Isgrò, and R. Prevete. Don't push the button! exploring data leakage risks in machine learning and transfer learning. *arXiv preprint arXiv:2401.13796*, 2024.

[3] M. A. A. Babu, S. K. Pandey, D. Durisic, A. C. Koppisetty, and M. Staron. Improving image data leakage detection in automotive software. *arXiv preprint arXiv:2410.23312*, 2024.

[4] M. A. Bouke and A. Abdullah. An empirical study of pattern leakage impact during data preprocessing on machine learning-based intrusion detection models reliability. *Expert Systems with Applications*, 230:120715, 2023.

[5] A. Burkov. *Machine learning engineering*, volume 1. True Positive Incorporated Montreal, QC, Canada, 2020.

[6] S. Chorev, P. Tannor, D. B. Israel, N. Bressler, I. Gabbay, N. Hutnik, J. Liberman, M. Perlmutter, Y. Romanyshyn, and L. Rokach. Deepchecks: A library for testing and validating machine learning models and data. *Journal of Machine Learning Research*, 23(285):1–6, 2022.

[7] F. Drobnjaković, P. Subotić, and C. Urban. An abstract interpretation-based data leakage static analysis. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 109–126. Springer, 2024.

[8] W. Epperson, A. Y. Wang, R. DeLine, and S. M. Drucker. Strategies for reuse and sharing among data scientists in software teams. In *Proceedings of the 44th international conference on software engineering: Software engineering in practice*, pages 243–252, 2022.

[9] G. Hulten. *Building Intelligent Systems: A Guide to Machine Learning Engineering*. Apress, 2018.

[10] S. Kaufman, S. Rosset, C. Perlich, and O. Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–21, 2012.

[11] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–11, 2018.

[12] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. In *Guide to advanced empirical software engineering*, pages 63–92. Springer, 2008.

[13] A. P. Koenzen, N. A. Ernst, and M.-A. D. Storey. Code duplication and reuse in jupyter notebooks. In *2020 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, pages 1–9. IEEE, 2020.

[14] Y. Liu, S. Mechtaev, P. Subotić, and A. Roychoudhury. Program repair guided by datalog-defined static analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1216–1228, 2023.

[15] T. M. Mitchell. *Machine learning (1 ed)*. McGraw-Hill, Inc., USA, 1997.

[16] N. Nahar, S. Zhou, G. Lewis, and C. Kästner. Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process. In *Proceedings of the 44th international conference on software engineering*, pages 413–425, 2022.

[17] H. Shimakawa, A. Kumada, and M. Sato. Prevention of leakage in machine learning prediction for polymer composite properties. *Journal of Chemical Information and Modeling*, 64(9):3621–3629, 2024.

[18] P. Subotić, L. Milikić, and M. Stojić. A static analysis framework for data science notebooks. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 13–22, 2022.

[19] A. P. S. Venkatesh, J. Wang, L. Li, and E. Bodden. Enhancing comprehension and navigation in jupyter notebooks with static analysis. In *2023 IEEE international Conference on software analysis, evolution and reengineering (SANER)*, pages 391–401. IEEE, 2023.

[20] A. Y. Wang, D. Wang, J. Drozdal, M. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan. Documentation matters: Human-centered ai system to assist data science code documentation in computational notebooks. *ACM Transactions on Computer-Human Interaction*, 29(2):1–33, 2022.

[21] C. Yang, R. A. Brower-Sinning, G. Lewis, and C. Kästner. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.