Workflow analysis of data science code in public GitHub repositories

Dhivyabharathi Ramasamy¹ · Cristina Sarasua¹ · Alberto Bacchelli¹ · Abraham Bernstein¹

Accepted: 8 August 2022 / Published online: 19 November 2022 © The Author(s) 2022

Abstract
Despite the ubiquity of data science, we are far from rigorously understanding how coding in data science is performed. Even though the scientific literature has hinted at the iterative and explorative nature of data science coding, we need further empirical evidence to understand this practice and its workflows in detail. Such understanding is critical to recognise the needs of data scientists and, for instance, inform tooling support. To obtain a deeper understanding of the iterative and explorative nature of data science coding, we analysed 470 Jupyter notebooks publicly available in GitHub repositories. We focused on the extent to which data scientists transition between different types of data science activities, or steps (such as data preprocessing and modelling), as well as the frequency and co-occurrence of such transitions. For our analysis, we developed a dataset with the help of five data science experts, who manually annotated the data science steps for each code cell within the aforementioned 470 notebooks. Using the first-order Markov chain model, we extracted the transitions and analysed the transition probabilities between the different steps. In addition to providing deeper insights into the implementation practices of data science coding, our results provide evidence that the steps in a data science workflow are indeed iterative and reveal specific patterns. We also evaluated the use of the annotated dataset to train machine-learning classifiers to predict the data science step(s) of a given code cell. We investigate the representativeness of the classification by comparing the workflow analysis applied to (a) the predicted data set and (b) the data set labelled by experts, finding an F1-score of about 71% for the 10-class data science step prediction problem.

Keywords Data science · Data science workflow · Data science workflow · Workflow analysis · Data science life cycle · Source code classification · Notebooks · Jupyter notebooks

Communicated by: Xin Xia

Dhivyabharathi Ramasamy
ramasamy@ifi.uzh.ch

Extended author information available on the last page of the article.
1 Introduction

Data science (DS) is the process of deriving insights from data (Muller et al. 2019b; Kery et al. 2017; Kim et al. 2016). Due to the continuously growing availability of vast amounts of data and computational capabilities, data science has been adopted across various fields, from drug design to software engineering (Menzies et al. 2016). Being able to properly do data science has become an important skill in today’s world (O’Neil and Schutt 2013; Miller and Hughes D 2017; PriceWaterhouseCoopers 2017). As a result, understanding how to assist data scientists based on their needs is a research area that has gained attention in recent years. While part of the literature focused on conducting interviews and surveys that reveal how data scientists actually work (Wang et al. 2019a; Zhang et al. 2020a; Wang et al. 2019a; Muller et al. 2019b; Wang et al. 2019b; Kery et al. 2018; Kery et al. 2017; Mao et al. 2019; Liu et al. 2020), other works provided technological solutions to improve the way data scientists work in terms of, for instance, collaboration (Patterson et al. 2017), code interaction (Rule et al. 2018; Pimentel et al. 2019; Kery et al. 2019), and automation (Guo and Seltzer 2012; Heffetz et al. 2020; Aggarwal et al. 2019). Our work, instead, provides an empirical investigation of the workflows (i.e., sequence of data science steps) implemented in real-world notebooks that were published on the Web.

A set of studies (Wang et al. 2019a; Zhang et al. 2020a; Aggarwal et al. 2019; Muller et al. 2019b) suggests that a data science workflow is comprised of multiple steps: data acquisition and preparation, preprocessing, feature engineering, modelling, visualisation, and deployment. Low-code data mining systems like RapidMiner (Hofmann and Klinkenberg 2016) and KNIME (Berthold et al. 2009) also group data mining and machine learning procedures and operators in terms of data science steps (data loading and transformation, data preprocessing and visualization, modelling, evaluation, and deployment). A high-level characterisation of such data-oriented steps is provided by Garijo et al. (2013a) through their empirical study of workflows developed in low-code (GUI-based) systems like Wings and Taverna. Such characterisation could help data scientists to reason about and orient themselves in a large collection of data science operators (Garijo et al. 2013a), as well as make sense of existing workflows (Garijo et al. 2013a). However, there is limited work on the empirical characterisation and analysis of workflows developed through code-based workflow development systems like Jupyter. We argue that extending this characterisation of different data science steps for such systems can provide various advantages.

First, it can inform the design of tools and interfaces to support data scientists. Data scientists generally prefer GUIs (Olabarriaga et al. 2014) and need tools similar to an IDE (Integrated Development Environment) (Chattopadhyay et al. 2020). For instance, in traditional IDEs, annotations have proven to be effective in aiding navigation and understandability (Chattopadhyay et al. 2020; Storey et al. 2009). Similarly, data science annotations could provide semantic cues and ease navigation difficulties (Kery et al. 2017) and help users shorten development and debugging time (Olabarriaga et al. 2014). They can help find relevant information (similar to ‘code browser’ in Eclipse) and allow developers to interact with their code easily (McCormick and De Volder 2004).

Second, aiding the analysis of data science workflows can provide insights for workflow development support. Workflow analysis has been well explored in traditional software engineering (SE) (Smith et al. 2005; Meena et al. 2005; Trcka et al. 2008; Colombo et al. 2006; Keith and Vega 2016). Rubin et al. (2007) suggests that improvements to the software process should be aided by the understanding of “what was actually done during the software development process and not by what is simply said about it” (Rubin et al.
2007, p. 170). It seems reasonable that this concept should be adapted to data science workflows as well. Past research studies data-intensive workflows and how they can be supported by low code workflow systems (Garijo et al. 2013a). Using annotations of data science activities in a workflow, these systems are expected to help design tools that allow users to modify, execute, debug, and reuse the (sub-)workflows (Carvalho et al. 2018; Garijo et al. 2013a; Krämer et al. 2013; LaToza and Myers 2010; Carvalho et al. 2017) (e.g., call graph).

Third, better understanding the data science workflows in practice is essential to devise automation and tools that better target data scientists’ needs (Garijo et al. 2013a; Chattopadhyay et al. 2020; Olabarriaga et al. 2014; Storey et al. 2009). For instance, they can provide active support to data scientists by automatically identifying the current step in the workflow under development and recommending the next step and its relevant (alternative) methods. Such identification enables dynamic modelling of both the data science step and its action space compared to the existing systems where the data science step is fixed (Heffetz et al. 2020). Such an AutoML system may incur less computational cost (Heffetz et al. 2020). They can also inform the design of AutoML systems that work as a partner and in collaboration with human practitioners (Wang et al. 2019a; Hernández-Orallo and Vold 2019; Zheng et al. 2017). For example, data scientists work on different steps of the workflow while collaborating in teams (Wang et al. 2019a). Through appropriate interventions (Muir 1994; Norman 1990; Parasuraman et al. 2000) in those steps where humans need help and (or) by automating them, AutoML systems can support a data scientist’s creation and management of data science workflows. Enabling such support requires identification of the steps in a data science workflow (Garijo et al. 2013a; Bennett et al. 2016) and the transitions between them.

In our work, we propose a way to characterise and identify different steps in data science workflows developed through code-based workflow systems like Jupyter notebooks. To that end, we develop an expert-annotated dataset and investigate supervised methods to extend the dataset by classifying the data science code automatically.

Furthermore, while the movement of data through a data science workflow typically hints at a waterfall-like pipeline, a few studies suggest that — in practice — data science may be more of an explorative process (Rule et al. 2018) that is iterative (Kery et al. 2019; Rule et al. 2018; Kandel et al. 2012a; Guo and Seltzer 2012) and non-linear (Kery et al. 2017; Rule et al. 2018). These studies provide limited empirical evidence: Rule et al. (2018) use the presence of non-linear execution numbers in a notebook as evidence of non-linearity. However, an execution number may simply reflect code-level changes and not necessarily a step execution. Another empirical study by Garijo et al. (2013a) reports the frequencies of data-related operations in workflows created through low-code workflow systems across domains. However, the study does not discuss the interaction between these operations.

In this paper, we provide further empirical evidence for the non-linearity and elaborate on the interactive patterns in these workflows. To this end, we analyse data science workflows implemented by data scientists to deepen our understanding of the nature of data science coding and the interactions within its steps. Specifically, we investigate which steps are frequently appearing in a data science workflow, whether they co-occur with other step(s), and what kind of interactions happen among them. We provide additional insights into each step of a data science workflow by analysing the number of lines of code and code clones in them. We also aim to further enable a large analysis of workflows in notebooks by creating an expert-annotated dataset of data science workflows and investigating the feasibility of supervised classification methods to expand the dataset.
The objects of our analysis are publicly available data science notebooks implemented in Jupyter. Jupyter is a popular development environment for data science tasks, which allows data scientists to create and share notebooks that contain a sequence of cells. The cells allow the logical separation of code chunks and may include live code, graphical representations of its results, as well as markdown text with equations and other natural language elements like explanations of design decisions, interpretations of results, and links to external documentation. These contextual elements intertwined with the source code contribute to the computational narrative (Jupyter 2015) of the notebook.

To conduct our analysis on these data science notebooks, first, we create a novel dataset (DASWOW – Data Science Notebooks with Workflow Information) with the contribution of data scientists. These data scientists manually annotated 470 data science notebooks (selected from the ~1M notebooks dataset provided by Rule et al. (2018)), containing 9,678 code cells, by labelling each code cell with the data science step(s) performed within. Second, we use Markov chain modelling to perform a workflow analysis on the dataset to uncover the data science implementation practices. Finally, we evaluate and compare the potential of supervised classification techniques in extending the dataset for large-scale workflow analysis. To train the classifier, we use automatically generated features for each data point (‘code cell’) in the DASWOW along with its corresponding label indicating the data science step it performs.

The main contributions of our study include:

– A novel, publicly available dataset, DASWOW, to support workflow analysis and report of a set of descriptive statistics related to code cells, variables, functions lines of code, and comment for the dataset.
– New empirical insights on data science workflows. We find that 29.75% of the code cells are written to explore the data — based on the primary purpose of the cell. In line with the common knowledge (Wang et al. 2019a), we also find that 23.5% of the code cells created by data scientists pre-process data. Moreover, 24% of the code cells in Jupyter notebooks implemented more than one type of data science activity, thus indicating the (1) overlapping nature of data science activities and (2) data scientists not using Jupyter cell structure to logically separate the data science activities. Our workflow analysis also shows that data science is iterative throughout and at every step in the workflow.
– Evidence that DASWOW can be extended with the help of existing supervised classification methods. For this, we employ and evaluate single-label and multi-label classification methods; we show that multi-label random forest classifier performs better in terms of F1-score when trained with features composed of lines of code and additional software metric information. We also discuss factors that might impact the performance of the classifier. Furthermore, we repeat the workflow analysis on the predicted dataset and show the representativeness of the classification.

The remainder of this article is organised as follows: Section 2 provides an overview of the relevant related work and lays out the research questions we investigate. Section 3 describes the creation of the expert annotated dataset — DASWOW. Section 4 presents the workflow analysis on DASWOW and details the insights we obtained. Section 5 shows the potential of supervised classification methods in extending the dataset and further discusses the evaluation results. We also report the workflow analysis on the predicted dataset and showcase its representativeness. Section 8 concludes the article with a brief summary of the insights and a discussion on the results’ potential.
2 Background and Research Questions

We provide an overview of the related work that leads to the main research questions (RQs) addressed in this article. Specifically, we look at the literature on data science workflows and their classification.

2.1 Characterisation of Data Science Workflows

A data science workflow consists of data-related activities that range from obtaining the data to modelling or interpreting it. Different studies consider that a data science workflow includes (some of) the following steps: data acquisition and preparation, preprocessing, feature engineering, modelling, visualisation, and deployment (Wang et al. 2019a; Zhang et al. 2020a; Aggarwal et al. 2019; Muller et al. 2019b; Souza et al. 2020). Garijo et al. (2013a) identified a similar set of data-related activities in the case of scientific workflows. In addition to the above, grey literature also includes other activities like business understanding/problem framing, data exploration, and reporting (Springboard 2016; Microsoft 2020; UCSD 2021). Other studies group the list of data science steps into a set of high-level tasks. For example, Kandel et al. (2012b) suggests the existence of the following tasks: discovery, wrangling, profiling, modelling, and reporting, whereas Kun et al. (2018) proposes questions, data, methods, inference, and design.

Given that computational notebooks are popular tools for data science implementations, a large-scale analysis of how these data science steps are implemented in existing notebooks can support understanding the data science workflows in practice. This requires a data set that contains expert annotations of data science workflows in data science notebooks. Zhang et al. (2020b) produced a dataset of 100 notebooks (randomly selected from Rule et al. (2018)) by assigning a label from a set of high-level data analysis task categorisations — ‘IMPORT, WRANGLE, EXPLORE, MODEL, EVALUATE’ — per cell.

However, the existing task categorisations do not represent the data science workflows in notebooks exhaustively. For example, some code fragments are also used to support the analysis in the form of helper functions (e.g., `import pandas`), to aid data loading, and visualisation. Many notebooks also have code fragments that support saving the obtained results. Therefore, to facilitate an empirical investigation of data science workflows in notebooks, this paper identifies the relevant steps in a data science workflow from the literature and extends them to computational notebook platforms. We provide annotations that contain the steps mentioned above and include the steps that are identified through literature in order to provide a complete characterisation of data science workflows in notebooks. We then use these steps as labels to manually annotate the notebook cells resulting in our first contribution: the DASWOW dataset. As we will outline in Sections 3 and 4, this approach allows us to better understand the inner workings of data science analyses.

2.2 Analysis of Data Science Workflows

The analysis of software development workflows and processes has been explored in the field of SE by several studies (Smith et al. 2005; Meena et al. 2005; Trcka et al. 2008; Colombo et al. 2006; Keith and Vega 2016; Rubin et al. 2007; Chan and Leung 1997).

Researchers, particularly from HCI and CSCW community, have explored data science practices (Zhang et al. 2020a; Kross and Guo 2019; Mao et al. 2019; Muller et al. 2019b; Passi and Jackson 2018; Rule et al. 2018; Wang et al. 2021a) and how data scientists work
with computational notebooks like Jupyter (Kery et al. 2019; Muller et al. 2019b; Kery et al. 2018; Kery et al. 2017; Rule et al. 2018; Chattopadhyay et al. 2020), primarily through surveys or expert interviews of data scientists. These studies are focused on understanding data science practices and workflows with a human-centric approach (Aragon et al. 2016; Muller et al. 2019a). Particularly, Kross and Guo (2019) found that instructors of data science emphasised the creation of computational workflows that center on ‘code, data, and communication’ during their teaching. Whereas, Zhang et al. (2020a) studied how data science teams collaborate through several stages of a data science workflow. In this work, we perform a quantitative study of workflows by characterising and empirically analysing the resulting artefacts of a data science process i.e., computational notebooks. As a result, we view our work to be complementary in enhancing our understanding of data science workflows in practice.

Previous research on scientific workflow analysis (Garijo et al. 2013a; Garijo et al. 2013b; Liu et al. 2015) focuses on workflows developed through low code systems like Taverna (Missier et al. 2010) and Wings (Gil et al. 2010) that allow the creation of scientific workflows through drag and drop interface and involve less or no coding. Studies on such workflows are concerned with issues like the management of these scientific workflows to provenance tracking in order to create reproducible and reusable workflows. One such study by Garijo et al. (2013a) focused on abstracting the data-intensive scientific workflows to identify the different types of tasks carried out in the workflows. Such abstractions can facilitate the understandability and reuse of workflows (Garijo et al. 2013a). In our work, we adapt and extend such characterisation for code-based computational notebook systems like Jupyter, further analysing the sequence in which data science steps (co-)occur.

There is currently no work\(^1\) that investigates the workflows developed through computational notebooks in order to understand the steps implemented in them, the details of their iterative nature and behaviour (interactions). While the literature documents the iterative and non-linear nature (Kery et al. 2019; Kandel et al. 2012a; Guo and Seltzer 2012; Kery et al. 2017; Rule et al. 2018) of data science notebooks, there is, yet, little empirical work on the analysis of workflow in data science notebooks. Rule et al. (2018) empirically analysed the iterative nature of Jupyter notebooks using the execution number available for each of the cells. Other studies focused on specific steps in the data science workflow (Lee et al. 2020; Hohman et al. 2020). Zhang et al. (2020b) analysed the role of individual steps in scientific data science workflows in academic publishing and how they differ between scientific domains. Lee et al. (2020) analysed workflows shared in the OpenML platform to extract information on the usage of machine learning (ML) techniques.

In our study, we focus on identifying whether some steps are more frequent in the workflow than others, with which other step(s) they frequently co-occur (if any), and how the steps interact with each other throughout the notebook. We also look at the distribution of lines of code across the workflow. Finally, given that anti-patterns such as code clones (Brown et al. 1998) are prevalent in computational notebooks (Koenzen et al. 2020; Källén and Wrigstad 2020) and are often investigated in software engineering analysis

\(^1\) to the best of our knowledge
(Rahman et al. 2012; Roy and Cordy 2007), we also investigate in what data science steps clones occur in a data science workflow. Specifically, we answer the following questions:

| RQ-1a | What are the most/least frequent data science steps implemented in notebooks? |
|-------|--------------------------------------------------------------------------------|
| RQ-1b | What kind of transitions appear in data science workflows?                     |
| RQ-1c | What is the distribution of the lines of code across different steps in a data science workflow? |
| RQ-1d | Where do cloning occur in a data science workflow?                              |

Previous studies analysing software workflows have used several analysis techniques including (timed) Markov-models (Smith et al. 2005; Colombo et al. 2006), activity diagrams (Meena et al. 2005) and Petri-nets (Trcka et al. 2008; Rubin et al. 2007). As we model the data science steps as states, we use the Markov chain model in order to compute the transition probabilities between them, assuming that the prediction of the next step is influenced only by the current state.

2.3 Classification of Source Code

Methods for classifying source code for various purposes (e.g., identification of programming language, authorship, topic, and quality) have been studied extensively in the field of software engineering. Particularly, machine learning techniques have been evaluated for their performance in classifying source code. Several techniques like Max-Entropy Classifier (ME) (Zevin and Holzem 2017), Support Vector Machines (SVM) (Ugurel et al. 2002), Decision Trees (DT) (Knab et al. 2006; Barstad et al. 2014), K-Nearest Neighbour (KNN) (Barstad et al. 2014), and Naive Bayes (NB) (Barstad et al. 2014) have been explored for source code classification. Applications of these techniques include prediction of programming language (Zevin and Holzem 2017; Ugurel et al. 2002), documents’ topic or content (Ugurel et al. 2002; Bacchelli et al. 2012), defects (Knab et al. 2006; Pascarella et al. 2019), code comments (Pascarella and Bacchelli 2017), code authorship (Pellin 2000), and code quality (Barstad et al. 2014).

Several features including code metrics (Barstad et al. 2014), (lexical) tokens — code, header file names, comments, README etc. represented in terms of n-grams and/or phrases (Zevin and Holzem 2017; Ugurel et al. 2002), syntactical structures (Zevin and Holzem 2017; Pellin 2000), and coding style (Pellin 2000) have been investigated in source code classification problems. Bacchelli et al. (2012) use sequence capturing features to classify e-mail messages related to software development using features such as @@-lineBefore and @@-lineAfter to consider the preceding and subsequent lines of code to recognise the structure of patch or stack trace content.

In terms of research on the classification of data science steps in source code, specifically, based on the workflow information, Zhang et al. (2020b) proposed weakly supervised transformer-based architecture to classify and assign source code to high-level data analysis tasks. In this article, we present our implementation and evaluation of a supervised data science code classification method by formulating it as a topic classification problem. We also investigate two classification strategies: single-label (one label per cell) and multi-label classification strategy to account for the potential existence of multiple data science activities in one given cell. Furthermore, we use the insights from existing SE literature to guide
our feature and model selection for the classification. We structure our research to answer the following research questions:

| RQ-2. How do existing supervised machine learning methods perform in data science code classification? |
|---------------------------------------------------------------|
| RQ-2a. Do non-code artefacts like markdown or comments and code metrics improve classification accuracy? |
| RQ-2b. Does the feature set size have any effect on classification accuracy? |

### 3 DASWOW: a Labelled Data set of Data Science Notebooks with Workflow Information

As previously discussed (Section 2), there is currently no annotated dataset to conduct an analysis of workflows in data science notebooks. Therefore, to conduct our investigation, we created DASWOW, a dataset of publicly available data science notebooks from GitHub that are annotated with workflow information. To allow other researchers to continue in this line of work, we make DASWOW publicly available together with the analyses here: https://doi.org/10.5281/zenodo.5635475. In the rest of this section, we detail how we created this dataset.

#### 3.1 Notebook Selection Procedure

We based DASWOW on a corpus of about one million Jupyter notebooks retrieved from GitHub by the Design Lab team at UC San Diego, which contains both the repository contents (including the code and other associated files such as README.md files) as well as repository metadata information such as owner information (Rule et al. 2018). Python notebooks account for a large portion of the corpus (96.04%), although it also contains notebooks in Julia (1.19%) and R (1.16%). The notebooks are obtained from over 1,000 repositories that were generated for diverse purposes (e.g., data science tasks, homework submissions, and software development (Rule et al. 2018)).

As we focus on data science notebooks, we manually inspected several notebooks to select a subset of 500 notebooks based on the following conditions:

**Purpose:** The notebook was developed for a data science task and has at least a cell.

**Kernel language:** The notebook uses Python as their kernel language.

Among the different types of cells supported in Jupyter notebooks (i.e., Markdown, Code, Raw NBConvert, and Output), we focus on code cells (i.e., we regard each code cell as a data point) as they provide a separation of the code snippets. Also, to implement a particular data science step, usually more than a single line of code is needed. Code cells contain lines of code written in the language associated with the kernel (Python in our case) and, on execution, produce a list of outputs that are displayed in the Output cell. Code cells

---

2The first author of this paper randomly sampled a set of notebooks and identified the purpose of the task implemented in them by manual inspection to select a set of 500 notebooks
can also contain code comments and may be denoted with an execution_count which indicates the order in which it was executed.

### 3.2 Annotation Labels

As the first step to the annotation task, we identify the existing data science activities by looking at the literature and online resources that discuss data science practices and extend them to computational notebooks. As we only focus on programming activities, we ignore steps that occur outside of notebooks, such as understanding and framing the problem, data collection, and ETL (Extract-Transform-Load), which is done with specialised tools (Vassiliadis et al. 2009), and deployment (Kery et al. 2018). As a result, we identify the following steps as part of data science workflows in notebooks: data exploration, data preprocessing, modelling, evaluation, as well as prediction. In addition, we consider loading the data into the notebooks as a different step (load_data) in the workflow as we noticed that these steps are generally performed in separate cells. Similarly, cells that visualise the results (result_visualization) and save them (save_results) for later use are found in Jupyter notebooks. We also identify cells that are being used for importing libraries, setting environmental parameters etc., which usually happens at the beginning of the notebook — as helper_functions. Additionally, we also encountered code cells that contain only comments (either code comments or commented code denoted as comment_only). As a result of this process, we propose the ten labels as listed in Table 1 to represent the steps in a data science workflow in Jupyter

| Table 1 Data Science steps annotated in DASWOW |
|-----------------------------------------------|
| **Label**          | **Definition**                                                                 |
| helper_functions   | Code that is not directly related to the data science activity at hand, but provides useful scripting functions (e.g. importing or configuring libraries). |
| load_data          | The process of loading a dataset of any type (e.g., .csv, .pkl) into a Jupyter notebook environment. |
| data_preprocessing | The process of preparing the dataset(s) for the subsequent analysis. It includes tasks such as cleaning, instance selection, normalisation, data transformation, and feature selection. |
| data_exploration   | The process of inspecting the content and shape of a dataset to understand the nature and characteristics of the data. Note that it may involve the usage of visualisation techniques but differs in its purpose. |
| modelling          | The process of applying statistical models and learning-based algorithms to learn from sample data. |
| evaluation         | The process of assessing a model using one/Various evaluation metric(s) such as goodness of fit and accuracy. |
| prediction         | The process of applying a model trained on a set of data to other or newly arriving pieces of data to forecast new values. |
| result_visualization| The process of obtaining a graphical representation (e.g., tables, plots, graphs) of a/Several measurement(s) |
| save_results       | The process of serialising and storing the data. |
| comment_only       | Lines of comment including commented code. |
Table 2 We showcase the proposed labels against categorisations from three different lines of work: computational notebooks, scientific workflows, and data science practices

| Proposed labels for Computational Notebooks | Corresponding labels identified from the literature |
|-------------------------------------------|---------------------------------------------------|
| Helper Functions                           | Computational Notebooks                            |
| Load Data                                  | Scientific Workflows                               |
| Data Preprocessing                         | Data Science Practices                             |
| Data Exploration                           | Zhang et al. [95]                                  |
| Wrangle                                    | Garjo et al. [49]                                  |
| Explore                                    | Wang et al. [88], Agrawal et al. [12]              |
| Modelling                                  | Model                                              |
| Evaluation                                 | Data Analysis                                      |
| Prediction                                 | Model Building and Training                        |
| Result Visualization                       | Hypothesis Optimization                            |
| Save Results                               | Ensemble                                           |
| Comment Only                              | Model Validation                                   |

We consider those labels that correspond to coding activities. Steps taken before the data is loaded into the notebooks (e.g., Data Acquisition) and steps that may occur outside notebooks (e.g., Deployment) are not included.

During the annotation activity, each code cell in the notebook is assigned a primary label — the main activity performed in the cell — along with other relevant labels that we define as secondary labels based on their presence. For example, if a cell performs modelling activity but also visualises the generated model within the same cell, then, we identify modelling as the primary label and result visualization as a secondary label. The detailed set of instructions used for the annotation task is provided in the Appendix (Section Appendix).

### 3.3 Expert DASWOW Annotation

Five data science experts (E1*, E2#, E3#, E4†, E5†) 4 annotated the selected 500 notebooks according to the following procedure. We conducted two pilot studies: one, to improve the annotation task setup and two, to hire external annotators with data science expertise in order to ensure consistent and quality labelling during the main annotation activity.

First, we conducted a pilot study with three experts (E1, E2, E3) who annotated a preliminary set of 50 notebooks based on an initial set of instructions in order to assess the clarity of the instructions and quality of the annotations. We used Cohen’s kappa (Cohen 1960) to assess the inter-annotator agreement between the expert’s annotations (all the relevant labels including primary label) for the 50 notebooks. Table 3 lists Cohen’s kappa for the different labels (i.e., data science steps), comparing the performance of E1 with E2 and E3. Based on the interpretation by Altman (1990), most of the agreements are between ‘moderate’ and ‘good’. Annotators saw data preprocessing in some instances as an extra step and thus, wrongly ignored it during the labelling leading to a ‘low’ agreement. Similarly,
Table 3  Annotator agreements from the first pilot task

| Label                    | E1 vs. E2 | E1 vs. E3 |
|--------------------------|-----------|-----------|
| helper_functions         | 0.624     | 0.449     |
| load_data                | 0.660     | 0.771     |
| data_exploration         | 0.590     | 0.562     |
| data_preprocessing       | 0.375     | 0.365     |
| modelling                | 0.524     | 0.620     |
| evaluation               | 0.493     | 0.588     |
| prediction               | 0.530     | 0.723     |
| result_visualization     | 0.674     | 0.433     |
| save_results             | 0.325     | 0.279     |
| comment_only             | **0.854** | **1.000** |

Cohens kappa score for the DS step annotation task. (0-0.2: Poor, 0.2-0.4: Fair, 0.4-0.6: Moderate, 0.6-0.8: Good, **0.8-1**: Very good). Bold numbers in the table indicate the top score, and numbers in coloured background cells indicate poor/fair scorers.

save_results had only a ‘fair’ agreement (highlighted rows in the table) as it occurred along with other steps and was seen as incidental rather than as an activity on its own. Another disagreement was that data visualisations that were a result of exploration activity were sometimes classified as result_visualization while we expected annotators to classify them as data_exploration according to the instructions. A different case of disagreement arose when annotators did not distinguish specific labels and considered them sub-set of other labels despite the instructions. For example, some annotators labelled evaluation and prediction actions as modelling even when corresponding distinct labels were available. In most cases, the disagreement was primarily driven by a misunderstanding or a subjective understanding of the definition, misreading the purpose of the code\textsuperscript{5}, and human error. After holding discussions among E1, E2, and E3, we resolved the disagreements by i) annotating a visualisation as data_exploration when it is a result of exploration activity, ii) emphasising model evaluation and model prediction activities with their corresponding labels, and iii) clarifying the details explicitly by revising the instructions with examples.

As a result of the preliminary annotation task, we improved the instructions (available in Section Appendix). We also incorporated the feedback from the experts to revise the annotations of the preliminary set of 50 notebooks (done by E1). We conducted a second pilot study in order to select further annotators for the annotation task, for which we invited five Upwork freelance data scientists selected based on their experience with data science tasks. They were asked to annotate 30 randomly selected notebooks of the preliminary set. We manually checked the quality of these annotations and also used Cohen’s kappa measure (refer to Table 4) to evaluate the annotations of each of the participants against E1’s annotations (which were updated based on the discussion with E2 and E3 as described above). Based on the results, we hired two of the five experts (E4 and E5) for the rest of the annotation process.

As a next step, E1 separately annotated further 50 notebooks. E4 and E5 annotated 200 notebooks each independently, taking the total to 500 notebooks (together with the preliminary set of 50 notebooks). As a result, each of the 500 notebooks is annotated by one of the experts. From the 500 notebooks we labelled in total, we retained notebooks with at least one of the following data science types of actions: data pre-processing, data exploration,

\textsuperscript{5}potentially due to lack of comments
modelling, and result visualisation. At the end of this process, we were left with 470 data science Python notebooks that we considered our object data for further analysis.

### 3.4 DASWOW Descriptive Statistics

DASWOW contains data on 470 notebooks containing 9,678 code cells with a total of 59,128 lines of code and 9,004 lines of comments. Figure 1 shows the distribution of the number of lines of code per cell in the dataset. The majority of the cells (~70%) have 1–5 lines of code. Table 5 provides further descriptive statistics for DASWOW. On average, a notebook has around 20 code cells and 125 lines of code; lines of comment are comparatively fewer (mean = 19), which is in line with observations from other studies (Rule et al. 2018). The statistics also reveal a substantial use of variables, whereas functions are scarcely used.

### 4 Workflow Analysis of Data Science Code

One of the central questions when investigating how to best support data scientists in their analysis is to understand the progression of the analysis that they go through. That is, we need to better understand the progression of higher-level activities — the data science steps

---

**Table 4** Annotator agreements from the second pilot task

| Label                | E1 vs. E4 | E1 vs. E5 |
|----------------------|-----------|-----------|
| helper_functions     | 0.514     | 0.899     |
| load_data            | 0.834     | 0.839     |
| data_exploration     | 0.600     | 0.591     |
| data_preprocessing   | 0.329     | 0.509     |
| modelling            | 0.573     | 0.553     |
| evaluation           | 0.641     | 0.513     |
| prediction           | 0.431     | 0.459     |
| result_visualization | 0.547     | 0.671     |
| save_results         | 0.430     | 0.262     |
| comment_only         | 1.000     | 0.969     |

Cohen’s kappa score for the DS step annotation task. (0-0.2: Poor, 0.2-0.4: Fair, 0.4-0.6: Moderate, 0.6-0.8: Good, 0.8-1: Very good). Bold numbers in the table indicate the top score, and numbers in coloured background cells indicate poor/fair scorers.
Table 5  DASWOW: Notebook-level statistics for 470 notebooks

|                  | no. of code cells | no. of variables | no. of functions | no. of lines of comment | no. of lines of code |
|------------------|-------------------|------------------|------------------|-------------------------|----------------------|
| mean             | 20.591            | 64.936           | 3.034            | 19.157                  | 125.804              |
| std              | 18.458            | 77.246           | 5.675            | 37.004                  | 138.732              |
| min              | 1                 | 1                | 0                | 0                       | 4                    |
| 25%              | 9                 | 23               | 0                | 2                       | 49                   |
| 50%              | 15.5              | 42.5             | 1                | 10                      | 82                   |
| 75%              | 26                | 74               | 4                | 23                      | 142.75               |
| max              | 162               | 732              | 46               | 523                     | 1052                 |

—and their inter-relationships. As a result, the remainder of this section is organised around a workflow analysis addressing research questions (Section 2.2) referring to the frequency in which the different types of data science steps occur across notebooks, the way these types of steps co-occur in code cells, and transitions between them.

4.1 RQ-1a. What are the Most/Least Frequent Data Science Steps Implemented in Notebooks?

Methodology: We compute the frequency distribution of the annotation labels that indicate the data science step(s) in each code cell of the Jupyter notebooks in DASWOW.

Results: When we consider that cells have only primary label (assuming cells have a main purpose i.e., to perform a single data science step), we find that, around 23.5% of the cells perform data preprocessing and around 30% of the cells perform data exploration (Fig. 2 shows the composition of all the data science activities in the DASWOW).

When we acknowledge that cells can refer to multiple data science steps, having a primary and a secondary step/label, the distribution slightly changes, as shown in Fig. 3. In the multi-label case, the order of evaluation (6.6%) and result visualization (4.3%) is flipped compared to the single label case (with 3.68% and 5.05% respectively). This increased occurrence of result visualization as a secondary label could indicate that, after a certain activity, results are usually visualised. Apart from this, we notice that helper functions is one of the most frequently appearing labels, which indicates data scientists import many external libraries into their notebooks.

![Fig. 2 Percentage of cells per label: Primary label](image)

Empir Software Eng (2023) 28:7  Page 13 of 47  7
We also distinguish between notebooks that contain a modelling label and notebooks that do not contain a modelling label to understand the composition of data science steps in modelling tasks. Particularly, we want to check whether data exploration is still prominent in modelling tasks. We compute their distribution of labels to understand the frequency of data science activities when a certain type of data science task (i.e., modelling or exploration) is performed.

The results (refer to Fig. 4) show that when notebooks include the modelling step, data exploration and data preprocessing account for 42.9% of the labels. In the case of notebooks that do not include the modelling step, data exploration and data preprocessing account for 77.14% of the total labels. The results show that result visualization is more frequent in notebooks that include modelling.

---

**Fig. 3** Percentage of cells per label: All labels

**Fig. 4** Percentage of cells per label by type of data science task: All labels. Modelling tasks perform data modelling and, in many cases, focused on predicting future outcomes. Non-modelling tasks do not model data and are instead focused on processing, exploring, or analysing the data.
As expected, evaluation and prediction appear only in those notebooks that contain modelling step. The interesting finding is that data exploration is consistently the most frequent data science step in notebooks whereas according to the literature, data preprocessing usually takes the majority of time (Muller et al. 2019b; Zhang et al. 2020a).

Furthermore, around 76% of the cells in our data set have one data science step, 19% have two steps, and 4.85% have more than two steps. In order to find out the frequently co-occurring labels within a single cell, we compute the frequency distribution of the label sets (all the labels per cell) in the dataset as the co-occurrence of certain data science steps or label sets (i.e., the presence of a certain combination of labels) could indicate a logical grouping of activities in the data science workflow. We observe co-occurrence for each possible label set. Table 6 shows the top-3 most frequently co-occurring label steps of sizes 1, 2, 3, and 4, i.e., labels that appear together in a cell but the cell may contain other labels as well. Most often (5.25%) co-occurring 2-label set is (data exploration, data preprocessing). This could indicate that exploration and pre-processing of data are activities that often occur in conjunction. Other frequently co-occurring data science label sets (excluding the general labels) with a share of at least 1% are (evaluation, modelling) at 4.27%, (modelling, prediction) at 1.95%, (evaluation, prediction) at 1.94%, (data exploration, modelling).

Table 6  Top-3 frequently appearing label combinations: a set of 1-label, 2-label, 3-label, and 4-label in the dataset

| Combination of Labels                             | % of co-occurrence | % appearance as a set |
|--------------------------------------------------|---------------------|-----------------------|
| 1-label                                          |                     |                       |
| (data exploration)                               | 39.37%              | 27.76%                |
| (data preprocessing)                             | 24.29%              | 16.74%                |
| (modelling)                                      | 18.24%              | 9.53%                 |
| 2-label                                          |                     |                       |
| (data exploration, data preprocessing)           | 5.25%               | 4.59%                 |
| (evaluation, modelling)                          | 4.27%               | 2.18%                 |
| (data exploration, load_data)                    | 3.3%                | 2.39%                 |
| 3-label                                          |                     |                       |
| (evaluation, modelling, prediction)              | 1.16%               | 0.53%                 |
| (evaluation, helper_functions, modelling)        | 0.85%               | 0.42%                 |
| (data exploration, helper_functions, load_data)  | 0.63%               | 0.52%                 |
| 4-label                                          |                     |                       |
| (evaluation, helper_functions, modelling, prediction) | 0.32%               | 0.27%                 |
| (evaluation, modelling, prediction, result_visualization) | 0.17%               | 0.13%                 |
| (evaluation, helper_functions, load_data, modelling) | 0.11%               | 0.08%                 |

% of co-occurrence indicates how frequent the combination of labels appears (may have other labels in the same cell) in the dataset. % appearance as a set indicates how frequent the combination of labels appears as a set (without other labels in the cell) in the dataset.
at 1.46%, and (evaluation, modelling, and prediction) at 1.16%. This indicates a logical grouping of activities in the data science workflow as expected, for example, similar to (Wang et al. 2019a).

We also obtain the top-3 most frequent label sets of sizes 1, 2, 3, and 4 in DASWOW, i.e., labels that appear as a set in a cell without other labels. data_exploration and data_preprocessing appear often (4.59%) as a label set. Other frequent data science label sets are (evaluation, modelling) at 2.18% and (data_exploration, load_data) at 2.39%.

We suspect another possible reason for the co-occurrence of data science steps is the lack of structure (Rule et al. 2018) and modularity (Desmond 2020) of data science code — resulting in large code chunks mixing various types of activities that are conducted in the same cell.

4.2 RQ-1b. What kind of Transitions Appear in Data Science Workflows?

Methodology: We compute the transition probabilities between types of steps using a first-order Markov chain model, where states represent the types of data science steps.

For each notebook, we set a start state and a stop state at the beginning and end of the notebook, respectively. Thus, the first cell has an incoming transition from start and the last cell in the notebook has an outgoing transition to stop. For example, a notebook with three cells will be mapped to:

start → load_data → data_preprocessing → modelling → stop

The corresponding set of transitions (each with a frequency of occurrence of 1) will be:

\[
\begin{align*}
\{ & \text{start} \rightarrow \text{load_data}, \text{load_data} \rightarrow \text{data_preprocessing}, \\
& \text{data_preprocessing} \rightarrow \text{modelling}, \text{modelling} \rightarrow \text{stop} \}
\end{align*}
\]

Once the transitions are identified, we compute the transition probability matrix. For cells that have been labelled with multiple labels (i.e., containing secondary labels), as we lack the information about the order of their appearance, we “naively” assign each edge in the transition equal probability using the formula: \(1/\text{no_of_edges}\).

Results: Figure 5 visualises the resulting transition probability matrix which reveals unique patterns of prominent iterations. Figure 6 visualises the transition matrix as a Sankey Flow Diagram (showing only transitions with probability > 0.05 for readability). The diagram shows that the data science workflows are iterative, not only at certain steps (e.g., modelling) but throughout its life-cycle. It reveals unique iterative patterns among and within different types of data science steps missing in the previous works, particularly within the following set of steps: (data_preprocessing, data_exploration), and (modelling, evaluation, prediction). More than 25% of the transitions starting at data_exploration flows into data_preprocessing and vice versa. Similarly, 22% of the transitions flow from modelling to evaluation, 38% from evaluation to modelling, 25% from prediction to modelling, and 27% from prediction to evaluation.

Furthermore, the matrix reveals that not only are there transitions between types of data science steps but also within types of steps (i.e., there are loops around the same step). The steps data_exploration, data_preprocessing, and modelling show that more than > 25% of the iterations take place within the same step type.
4.3 RQ-1c. What is the Distribution of the Lines of Code Across Different Steps in a Data Science Workflow?

**Methodology:** We compute the number of lines of code for each of the data science steps as primary label (per cell) in DASWOW.

**Results:** Figure 7 shows the results using a box plot. To draw our inference, we assume that number of lines of code can refer to the potential effort a data scientist must invest in implementing a particular data science step. We consider the assumption reasonable even when some of the lines of code are clones, given one should put effort into modifying and/or understanding the code clone to the task in hand (Kery et al. 2017). We find that, in general, modelling takes considerable effort as it contains a large number of lines of code. Although, if we consider the outliers, data preprocessing step has the highest number of lines of code written.

If we ignore the outliers, our dataset shows that the initial steps of load_data, data_preprocessing, and data_exploration together account for a similar amount of code written for modelling step. Furthermore, a large number of lines of code are written for result_visualization showing that data scientists consider the communication of results as an important activity. Apart from this, we notice that a large number of lines of code are written for helper functions which reiterates the finding (refer to Section 4.1) that data scientists import many external libraries into their notebooks.
Fig. 6 Transition matrix probabilities (> 0.05) visualised in a Sankey diagram. While modelling, evaluation and prediction is iterative as expected, the workflow throughout its life cycle shows iterative-ness

4.4 RQ-1d. Where do Cloning Occur in a Data Science Workflow?

Methodology: We compute the number of lines of code that is identically (i.e., type-1) cloned, appearing at least three or more times (by following the ‘rule-of-three’ (Neill et al. 2011)) within the same notebook in DASWOW. In order to understand the purpose of the
code that is being cloned, we use the primary label of the cell where every time the code appears. We use the number of appearances and primary label in conjunction, i.e., a clone appearing at least three times in the cell(s) with the same primary label. It is possible some of the clones may appear in cells that have a different primary purpose. We leave their detailed study and study on other types of clones\textsuperscript{6} to future work.

Results: We find in our dataset that \( \sim 51\% \) of the notebooks have at least one line of code cloned in them. Of the notebooks that have clones, eight (or more) lines of code are cloned on average per notebook, and \( \sim 26\% \) of the notebooks have clones higher than average. In Fig. 8, we show the percentage share of type-1 clones in DASWOW based on what data science step they appear in alongside the percentage share of notebooks that have clones in that data science step. Our results from data science notebooks are in line with the trend of a high occurrence of clones in Jupyter notebooks in general (Koenzen et al. 2020; Källén and Wrigstad 2020). Additionally, DASWOW allows us to understand the clones at a spatial level by situating them in the data science workflow.

We find that most of the type-1 clones (\( \sim 66\% \)) appear in modelling and data exploration step. A large number of clones in these steps could mean data scientists repeating or iterating over these steps, for example, executing a scikit model again after modifying the data. Our results also show that modelling and data exploration are also the most likely steps to be cloned in a notebook, as they appear in \( > 30\% \) of the notebooks in DASWOW.

Besides, our analysis shows a conservative account of clones in data science notebooks. We hypothesise that relaxing our conditions of identical-ness, a minimum number of appearances (three or more) in the same step, and within-notebook duplication could lead to a higher number of clones appearing in each step (e.g., import statements, scikit functions). Whilst code clones are usually seen as a bad practice in SE (Fowler 2018) for creating

\textsuperscript{6}This includes type-2 clones that have identical syntax but different user-defined identifiers, type-3 clones which are modified clones with added or deleted or modified statements, and type-4 clones that have different syntax performing the same computation (Koenzen et al. 2020)
overhead in maintainability and comprehensibility (Roy and Cordy 2007), there might be reasons why they may be acceptable or inevitable (Källén and Wrigstad 2020) in data science notebooks, for example, to simplify development effort (Kery et al. 2018; Koenzen et al. 2020). Hence, future work will have to investigate how to support and handle code clones, particularly in steps where they appear frequently.

4.5 Summary

Our workflow analysis provides insights that can support designing tools that assist data scientists while creating and managing their data science workflows. We discuss the implications of the findings in detail in Section 6. In order to further cater to various needs, a large-scale workflow analysis on data science notebooks based on different dimensions (like task type and domain) could provide further helpful insights for designing tools that support data science development. However, the process of manually labelling a dataset is expensive. Hence, in the next section, we investigate supervised machine learning techniques that can be used to implement such an extension of our data set by labelling the steps present in the cells automatically.

5 Extending DASWOW Using Supervised Machine-Learning

The workflow analysis presented in the previous section provided us with insights into the data science step a developer is engaged in. However, to derive further insights that can provide an important basis to both understand the data science workflow practices and provide appropriate tooling, large-scale analysis on an even larger data set of workflow information...
would be beneficial. As manual labelling does not scale to very large collections, in this section, we investigate to what extent DASWOW can be used to train a classifier capable of providing highly accurate data science step annotations. To that end, this section aims to answer ‘How well do current supervised machine learning methods perform when predicting data science steps in Python Jupyter Notebooks?’. First, we describe the preparation of the data, and then the methodology of developing the classifiers, followed by an evaluation of the classifiers’ performance. Subsequently, we provide an evaluation of the representativeness of the classifier results by comparing the workflow analysis based on manually annotated and predicted labels.

5.1 Methodology

We first describe our data preparation process which includes feature engineering and data pre-processing.

5.1.1 Feature Engineering

In addition to the data science step labels for the training data, we generated 15 features (lexical/textual and numerical) to represent a code cell. They refer to the content of code cells, their metadata (e.g., cell number), and information related to their neighbouring cells (e.g., markdown_heading). Given markdown cells can provide useful information, for each code cell, we consider the first line (Pimentel et al. 2019) of the preceding markdown as one of the features. We also considered the usage of repository information (such as forks, watcher, stars, owner information etc.) of a notebook; however, given their sparsity and initial insight into their informativeness, we decided not to use them further. Moreover, using an external library Radon, features such as number of distinct operators and operands, bugs, time and complexity, number of code cells in a notebook, number of markdown cells in a notebook, etc., were generated at the notebook level. However, we did not consider them for classification after an initial evaluation as the library requires the code to be represented as .py file, which results in the loss of its structural information. Instead, we generated code metrics relevant to each code cell using custom functions. The detailed list of features, its definitions and the rationale for its inclusion are listed in the Appendix (Section Appendix). The features are extracted directly from the explicit information in the notebook (e.g., execution count) or generated programmatically (e.g., number of variables). As a result, we have the following sets of features to represent each code cell:

- **Cell features**: are based on the content available in the cell itself, such as lines of code, comments, execution count as provided by its Jupyter environment etc., its meta-data and related components (such as output obtained from its execution).
- **External features**: indicate the man-page information about the libraries imported in a given cell which gives more information about the functions and use of a library.
- **Software metrics**: contain code metrics (e.g., number of variables) for a given cell using custom functions we implemented.

In total, each code cell in DASWOW is represented by 15 features.

---

7 Radon introduction to Code Metrics - https://radon.readthedocs.io/en/latest/intro.html. Available at https://pypi.org/project/radon/.
8 To generate the features programmatically, we implemented a parser to parse the notebook and its metadata.
5.1.2 Data Preprocessing

Before applying the classification model, we apply common preprocessing procedures to the features: where applicable, we apply `pandas.factorize()` to obtain the numerical representation (by one-hot encoding) of non-numerical features with a distinct set of values (e.g., output_type). For the lexical features like code, comments, and markdown heading, we first remove the stop words by setting the language to ‘English’ and punctuation. We then use `TfidfVectorizer`\(^9\) to tokenize the lexical features by extracting uni-, bi-, and tri-grams. We also enable inverse-document-frequency re-weighting. Using a trial, we set the threshold for document frequency at a maximum of 0.2% and a minimum of 2% to ignore corpus-specific stop words and infrequent words, respectively. As a result, lines of code of a cell are represented by a feature vector of size 48,485. When using numerical features, we apply standardisation\(^10\) to the features. We use the standard chi2 method for feature selection to select \(k\) most relevant code features (token) and discuss in detail the performance of classifiers over various sizes in Section 11.

5.1.3 Evaluating the Classification Methods

As each data point in our dataset contains one primary label and zero or more secondary labels, we investigate two relevant multi-class classification (more than two target classes) problems: single-label and multi-label\(^11,12\). Single-label classification assigns one label whereas multi-label classification assigns one to many labels per data point. For the single-label classification, we use the primary label as the classification target and predict one label per code cell. For the multi-label classification, we use all the applicable labels i.e., primary label and secondary labels as targets and, as a result, predict multiple labels per code cell. In the case of multi-label classification, we use the binary relevance approach implemented via the OnevsRest strategy. We formally define the classification of data science steps as follows.

Given a dataset \(D\) containing a set of samples \(\{X, y\}\), where \(X_i\) is a feature vector and \(y_i\) is its corresponding label with \(y_i \in \{1, ..., K\}\), a set of classifiers \(f_k(x | \theta_k)\) is learned, where \(\theta_k\) represents its corresponding optimal parameters. In our classification problem, \(K = 10\) (Table 1).

For an unlabelled sample \(x\), single-label classification learns the following decision function:

\[
\hat{y} = \arg \max_{k \in \{1, ... , 10\}} f_k(x) \tag{1}
\]

For the multi-label classification learns the following decision function:

\[
\hat{y} = \{y_1, y_2, \ldots, y_K\}, \text{ where } y_k = \begin{cases} 1 & \text{if } f_k(x) \geq t \\
0 & \text{otherwise} \end{cases} \text{ for } k \in \{1, \ldots, 10\} \tag{2}
\]

For the classification task\(^13\), we applied random sampling on DASWOW to create the training (60%), validation (20%) and test (20%) sets (Lever et al. 2016). To retain all the data points of a given notebook in a single set and as a result, to train the classifier with (latent) order information relevant to a notebook, the random sampling was executed on

\(^9\)https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
\(^10\)https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
\(^11\)using scikit’s default OnevsRest
\(^12\)https://scikit-learn.org/stable/modules/generated/sklearn.Multi-class.OneVsRestClassifier.html
\(^13\)For feature ablation study, we utilise the validation set as training samples as well.
filenames rather than data points (‘code cells’) themselves. As a result, the training, validation and test sets contain 282, 94, and 94 notebooks, respectively. The validation set is used in GridSearchCV\(^{14}\) for 10-fold cross-validation using scikit’s PredefinedSplit\(^ {15}\). The predefined split allows us to retain all cells belonging to a notebook together and at the same time perform n-fold cross-validation. The test set is used to evaluate the performance of the classifier.

**Classifiers** Given that the goal is to illustrate the performance that a classifier provides for data science predictions, we opted for comparing a set of well-known, off-the-shelf classifiers — an approach that can serve as a baseline for future explorations of supervised classifiers. Specifically, we investigated the performance of six classifiers: Random Forest (RF), Decision Tree (DT), Gradient Boosting (GB), Linear Support Vector (LSV), Support Vector (SV), and Logistic Regression (LOG). We use standard scikit-learn implementations for each of the classifiers.

In order to assess their usefulness, we create and evaluate the following six sets of feature combinations from the 15 features for each of the classifiers based on the (additional) information they provide while keeping code as the focal point:

1. **code** containing only the lines of code (represented by text in the DASWOW)
2. **code-comment** containing the lines of code and additional information from the comments (represented by comment in the DASWOW)
3. **code-stat** containing the lines of code and the additional software metric information (represented by linesofcomment, linesofcode, variable_count, function_count in the DASWOW) for a cell
4. **code-cell** containing the lines of code and other cell-related features
5. **all-features** containing all (code, comment, stat, and cell) the features.
6. **all-features-without-code** containing all the features excluding the lines of code.

**Classifier evaluation metrics** To evaluate the performance of the classifiers, we consider the \(F1\)-score\(^{16}\) (Tsoumakas and Vlahavas 2007), which is the harmonic mean calculated using precision and recall:

\[
\text{f}_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]

(3)

whereas precision and recall are derived from true positives (TP), true negatives (TN), and false positives (FP) as below:

\[
\text{precision} = \frac{TP}{TP + FP}
\]

(4)

\[
\text{recall} = \frac{TP}{TP + FN}
\]

(5)

\(^{14}\)GridSearchCV: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

\(^{15}\)PredefinedSplit: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.PredefinedSplit.html

\(^{16}\)Scikit’s F1-score implementation: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html
We also look at other most common evaluation metrics like accuracy\textsuperscript{17} for single-label classification, and subset accuracy, hamming loss,\textsuperscript{18} and jaccard similarity\textsuperscript{19} for multi-label classification (Park and Read 2018). Note that in multi-label classification, instead of accuracy, subset-accuracy (also called ‘Exact-Match’ accuracy), which indicates the percentage of samples that have all their labels classified correctly, is calculated. The Exact-Match accuracy is considered a harsh metric as it requires all of the labels for a given sample to be correctly predicted (i.e., if one of the labels in the set does not match, it is considered incorrect).

5.2 Results

We first provide a comparison of the performance of various classifiers for both single and multi-label classification. Then, we proceed to discuss the best-performing classifier in detail. We performed a detailed feature ablation study that takes into account both the combination of features and the code feature vector size over different classifiers, which we later discuss in this paper (refer to Section 5.2.3).

5.2.1 Predicting one Label per Code Cell

The accuracy and F1-score of all the classifiers evaluated over a different set of features are listed in Table 7. The size of the code feature vector is 1000 which is the top performing amongst all.

The RF classifier trained with code-stat features has the highest accuracy of 71.1%, closely followed (69.3%) by the GB classifier trained with all-features.

Best performing classifier We take the best performing classifier (RF) trained with code-stat for further analysis using GridSearchCV. The best performing parameters evaluated by GridSearchCV (with parameters n_estimators=90, class_weight=‘balanced_subsample’, random_state=500, criterion=’gini’) achieves a weighted average F1-score of 0.698.

Table 8 shows the classification report for the RF classifier.

Data preprocessing has a precision value of 0.74 and a recall value of 0.77. The reason for such a result could be that the lines of code that performs data preprocessing do not have many unique structures (e.g., plt.figure() in the case of result visualization or DecisionTreeRegressor() in the case of modelling). In the case of data exploration, even though it has a lower precision score, the recall score is higher indicating that many of the labels are predicted as data exploration even when they are not; the confusion matrix visualises this behaviour (Fig. 9).

Evaluation and prediction have lower recall values (< 0.30) and precision (< 0.55) values. We suspect this is because both these labels occur usually in conjunction with modelling (as found in our analysis (refer to Section 4.1)). Also, this label information is lost as we have used only primary label here. It is also possible that the presence of features relevant to other labels in a given data point might be a source of confusion for the classifier. For example, consider a code cell performing modelling as a main activity (captured

\textsuperscript{17}https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

\textsuperscript{18}https://scikit-learn.org/stable/modules/generated/sklearn.metrics.hamming_loss.html

\textsuperscript{19}https://scikit-learn.org/stable/modules/generated/sklearn.metrics.jaccard_similarity_score.html
Table 7 Single-label classification: Comparison of accuracy and F1-score metric of classifiers over a different set of features

| model | accuracy | F1-score |
|-------|----------|----------|
|      | code     |          |
| RF   | 0.639    | 0.622    |
| DT   | 0.547    | 0.562    |
| GB   | 0.627    | 0.611    |
| LSVC | 0.639    | 0.624    |
| SVC  | 0.626    | 0.608    |
| LOG  | 0.633    | 0.617    |
|      | code-comment |  |
| RF   | 0.624    | 0.610    |
| DT   | 0.505    | 0.526    |
| GB   | 0.620    | 0.608    |
| LSVC | 0.640    | 0.627    |
| SVC  | 0.633    | 0.619    |
| LOG  | 0.627    | 0.610    |
|      | code-stat |  |
| RF   | 0.711    | 0.703    |
| DT   | 0.571    | 0.596    |
| GB   | 0.693    | 0.683    |
| LSVC | 0.655    | 0.642    |
| SVC  | 0.660    | 0.649    |
| LOG  | 0.642    | 0.625    |
|      | code-cell-context |  |
| RF   | 0.604    | 0.586    |
| DT   | 0.383    | 0.424    |
| GB   | 0.582    | 0.572    |
| LSVC | 0.528    | 0.511    |
| SVC  | 0.518    | 0.477    |
| LOG  | 0.541    | 0.522    |
|      | all-features |  |
| RF   | 0.663    | 0.648    |
| DT   | 0.438    | 0.475    |
| GB   | 0.638    | 0.631    |
| LSVC | 0.554    | 0.540    |
| SVC  | 0.543    | 0.510    |
| LOG  | 0.550    | 0.535    |
|      | all-features-without-code |  |
| RF   | 0.616    | 0.589    |
| DT   | 0.339    | 0.373    |
| GB   | 0.597    | 0.572    |
| LSVC | 0.469    | 0.433    |
| SVC  | 0.430    | 0.361    |
| LOG  | 0.458    | 0.408    |

by primary label) that also performs evaluation in the later lines of code (captured by secondary labels). In this case, the single-label classifier will encounter features that might be relevant to both modelling and evaluation.
Table 8  Single-label classification: performance metrics of RF classifier

| labels                | precision | recall | F1-score | support |
|-----------------------|-----------|--------|----------|---------|
| helper_functions      | 0.82      | 0.79   | 0.81     | 154     |
| load_data             | 0.76      | 0.70   | 0.73     | 158     |
| data.preprocessing    | 0.74      | 0.77   | 0.75     | 591     |
| data.exploration      | 0.63      | 0.81   | 0.71     | 432     |
| modelling             | 0.69      | 0.65   | 0.67     | 316     |
| evaluation            | 0.49      | 0.28   | 0.36     | 95      |
| prediction            | 0.55      | 0.20   | 0.29     | 60      |
| result.visualization  | 0.88      | 0.50   | 0.64     | 30      |
| save_results          | 0.82      | 0.51   | 0.63     | 45      |
| comment_only          | 1.00      | 0.86   | 0.93     | 37      |
| accuracy              |           |        | 0.71     | 1918    |
| macro avg             | 0.74      | 0.61   | 0.65     | 1918    |
| weighted avg          | 0.71      | 0.71   | 0.70     | 1918    |

5.2.2 Predicting Multiple Labels per Code Cell

We compare the performance of classifiers over various sets of features for multi-label classification. The results are listed in Table 9. The size of the code feature vector is 2000 which is the top performing amongst all.

The RF (random forest) classifier trained with *code-stat* achieved the highest F1-score of 0.716 and a subset accuracy of 0.587. Since we use a total of ten classification labels in multi-label classification, we expected the subset accuracy to be less as it is a harsh metric. Table 9 shows that all of the classifiers have a hamming loss that is < 0.180. A low hamming loss score means that the proportion of labels predicted incorrectly is low. Jaccard similarity, which measures the similarity between the set of predicted labels and the corresponding set of true labels, is a midpoint between hamming loss and subset accuracy (Park and Read 2018) and is more likely to show a balanced view of the classifier’s performance. The RF classifier has a Jaccard similarity coefficient score of 0.570 i.e., the set of labels predicted is 57% similar to the set of true labels. The GB classifier trained with *code-stat* is the second-best performing with a Jaccard similarity score of 0.489 and an F1-score of 0.674.

Best performing classifier We take the best performing classifier (RF) trained with all-features for further analysis using GridSearchCV. The best performing parameters evaluated by GridSearchCV is Random Forest (with parameters n_estimators=100, class_weight='balanced_subsample', random_state=500, criterion='gini').

Table 10 shows the classification report for the RF classifier. Most of the classification labels have precision value $\geq 0.74$ except data.preprocessing (0.69) and result.visualization (0.43). In terms of recall, helper.functions, data.preprocessing, data.exploration, and comment_only have recall values equal to or greater than 0.75. The recall value for result.visualization has worsened more than double compared to the single-label classification indicating an
increase of false negatives. On the other hand, for result_visualization, the precision value has worsened along with the recall value indicating the classifier performs poorly with this label. While maintaining the weighted average of $F1$-score, multi-label classification has improved in average precision by 9.9% whereas degraded in recall by 5.6%. We also notice that the RF classifier did not assign any label to 6.4% of the data points (see Fig. 10 showing the comparison of the distribution of labels in the true vs. predicted set).

While this is a concern which is reflected in a low weighted average recall score, the classifier has a higher weighted average precision i.e., not many labels are inaccurately assigned.

### 5.2.3 Feature Ablation

**RQ-2a Do non-code artefacts like markdown or comments and code metrics improve classification accuracy?** For multi-label classification, the performance of the set of features follows the pattern:

$$F1(all\text{-}features\text{-}without\text{-}code) < F1(code\text{-}extend) < F1(all\text{-}features) = F1(code\text{-}comment) <= F1(code) < F1(code\text{-}stat)$$
Table 9  Comparison of a set of metrics of classifiers over a different set of features: multi-label classification

| model  | subset accuracy | hamming loss | jaccard similarity | F1-score |
|--------|-----------------|--------------|--------------------|---------|
|        | code            |              |                    |         |
| RF     | 0.448           | 0.078        | 0.495              | 0.648   |
| DT     | 0.363           | 0.097        | 0.443              | 0.602   |
| GB     | 0.400           | 0.079        | 0.465              | 0.615   |
| LSVC   | 0.429           | 0.080        | 0.482              | 0.638   |
| SVC    | 0.391           | 0.084        | 0.412              | 0.567   |
| LOG    | 0.407           | 0.083        | 0.451              | 0.611   |
|        | code-comment    |              |                    |         |
| RF     | 0.520           | 0.081        | 0.508              | 0.660   |
| DT     | 0.402           | 0.107        | 0.440              | 0.599   |
| GB     | 0.379           | 0.084        | 0.447              | 0.597   |
| LSVC   | 0.420           | 0.082        | 0.473              | 0.630   |
| SVC    | 0.397           | 0.084        | 0.420              | 0.577   |
| LOG    | 0.399           | 0.083        | 0.440              | 0.600   |
|        | code-stat       |              |                    |         |
| RF     | 0.587           | 0.068        | 0.570              | 0.716   |
| DT     | 0.455           | 0.096        | 0.487              | 0.647   |
| GB     | 0.489           | 0.072        | 0.523              | 0.674   |
| LSVC   | 0.428           | 0.081        | 0.481              | 0.638   |
| SVC    | 0.394           | 0.083        | 0.417              | 0.572   |
| LOG    | 0.411           | 0.082        | 0.456              | 0.616   |
|        | code-cell-context |          |                    |         |
| RF     | 0.495           | 0.078        | 0.495              | 0.640   |
| DT     | 0.256           | 0.147        | 0.373              | 0.532   |
| GB     | 0.434           | 0.090        | 0.474              | 0.617   |
| LSVC   | 0.403           | 0.098        | 0.429              | 0.583   |
| SVC    | 0.329           | 0.097        | 0.337              | 0.458   |
| LOG    | 0.391           | 0.096        | 0.415              | 0.565   |
|        | all-features    |              |                    |         |
| RF     | 0.536           | 0.072        | 0.528              | 0.672   |
| DT     | 0.306           | 0.140        | 0.403              | 0.561   |
| GB     | 0.522           | 0.076        | 0.531              | 0.680   |
| LSVC   | 0.406           | 0.097        | 0.434              | 0.590   |
| SVC    | 0.355           | 0.096        | 0.348              | 0.471   |
| LOG    | 0.393           | 0.095        | 0.423              | 0.574   |
|        | all-features-without-code |       |                    |         |
| RF     | 0.477           | 0.082        | 0.454              | 0.590   |
| DT     | 0.182           | 0.173        | 0.287              | 0.438   |
| GB     | 0.437           | 0.097        | 0.442              | 0.579   |
| LSVC   | 0.335           | 0.112        | 0.321              | 0.447   |
| SVC    | 0.321           | 0.111        | 0.262              | 0.353   |
| LOG    | 0.326           | 0.111        | 0.303              | 0.419   |

Hence, the answer to **RQ-2a** is that code features along with statistical information perform better in the classification of data science steps.

**RQ-2b Does the feature set size have any effect on classification accuracy?** Apart from investigating the effect of six sets of feature combinations: *code, code-comment, code-stat, code-cell, all-features, and all-features-without-code* on the classifier performance, we also looked into the effect of the size of the code feature (i.e., by selecting $k$ tokens using *chi2*) through an ablation study to answer our research questions.
Table 10  Multi-label classification: performance metrics of RF classifier

| labels                | precision | recall  | F1-score | support |
|-----------------------|-----------|---------|----------|---------|
| helper_functions      | 0.91      | 0.82    | 0.86     | 286     |
| load_data             | 0.88      | 0.66    | 0.75     | 191     |
| data_preprocessing    | 0.69      | 0.75    | 0.72     | 437     |
| data_exploration      | 0.80      | 0.79    | 0.79     | 801     |
| modelling             | 0.78      | 0.58    | 0.67     | 361     |
| evaluation            | 0.74      | 0.30    | 0.43     | 171     |
| prediction            | 0.77      | 0.33    | 0.47     | 90      |
| result_visualization  | 0.43      | 0.16    | 0.24     | 97      |
| save_results          | 0.84      | 0.53    | 0.65     | 30      |
| comment_only          | 1.00      | 0.86    | 0.93     | 37      |
| micro avg             | 0.78      | 0.67    | 0.72     | 2501    |
| macro avg             | 0.78      | 0.58    | 0.65     | 2501    |
| weighted avg          | 0.78      | 0.67    | 0.71     | 2501    |
| samples avg           | 0.74      | 0.71    | 0.71     | 2501    |

Results: Table 11 shows the performance over different sizes of feature vector. Based on the classifiers evaluated, we noticed that there is no clear pattern in the performance of the classifiers depending on the size of the code feature vector. While a code vector of size 5000 or 10000 performs better, increasing the size beyond 1000 does not seem to promise significantly better performance. As a result, the answer to RQ-2b is unclear. However, we found that the classifier trained with 470 notebooks (a larger set of data points) performs better than a subset of randomly selected 100 notebooks, which had an F1-score of 0.48.

5.2.4 Summary

The evaluation of classifiers in Section 5.2 showed the capability of existing supervised methods in extending DASWOW for large-scale workflow analysis on data science notebooks. Even though the classifier does not assign any label to 7% of the data points, the frequency distribution and transition probabilities of the predicted set show the effectiveness of the classifier. For a classifier to effectively extend the DASWOW dataset, it should produce zero false positives. False positives compromise the quality of the resulting dataset and, therefore, are costly. As an inaccurately labelled data point is worse than an unlabelled data point, this, in our case, can lead to an inaccurate workflow analysis. For this reason, a classifier that has a good F1-score and at the same time a better precision score is useful. In our evaluation of classifiers, multi-label classification with RF classifier has a weighted average F1-score of 0.71 and a weighted average precision of 0.78 compared to single-label classification with RF classifier’s weighted average F1-score of 0.70 and a weighted average precision of 0.71. Hence, in this case, multi-label classification is preferable.
5.3 Workflow Analysis-Based Evaluation

In this section, we evaluate the results of the classifier in terms of the workflow analysis. We apply the same methodology followed in Section 4 to the dataset containing the data science step labels predicted by the best performing multi-label RF classifier (see Section 5.2) and compare the workflow analysis results obtained in both cases (i.e., a dataset with manually obtained labels and dataset with predicted labels) in order to assess the potential of the supervised classification methods in extending the dataset to support large-scale analysis. We present these results for each of the three individual research questions:

**Frequency (RQ-1a)** We found that the distribution of the labels in our prediction set (see Fig. 11) is close to the distributions obtained using the training dataset except for a few labels. Particularly, there is a reduction in result_visualization. We computed Kendall’s rank correlation ($\tau$) and found that the samples from training and the predicted set of labels are highly correlated with a coefficient of 0.956. The result shows the effectiveness of the classifier.

Furthermore, the distribution of no of labels per cell in the predicted set has similar composition to the true composition, with around 76% of the cells in our data set having one data science step, 14% having two steps, and 2% having more than two steps. However, around 7% of the cells are unassigned, resulting in Kendall’s rank correlation co-efficient of 0.60 compared to the composition in DASWOW.

**Transitions (RQ-1b)** Transition probabilities for the predicted set of labels based on the equal strategy (refer to Section 4.2) are shown in Fig. 12.
Table 11  Multi-label classification: Comparison of *F1-score* over different set of features and sizes of the feature vector

| model  | 100 | 1,000 | 2,000 | 5,000 | 10,000 |
|--------|-----|-------|-------|-------|--------|
| **code**                                      |
| RF     | 0.507| 0.648 | 0.663 | 0.672 | 0.667  |
| DT     | 0.485| 0.602 | 0.612 | 0.621 | 0.631  |
| GB     | 0.525| 0.615 | 0.634 | 0.630 | 0.633  |
| LSVC   | 0.498| 0.638 | 0.646 | 0.651 | 0.654  |
| SVC    | 0.493| 0.567 | 0.589 | 0.597 | 0.585  |
| LOG    | 0.447| 0.611 | 0.644 | 0.655 | 0.665  |
| **code-comment**                              |
| RF     | 0.512| 0.660 | 0.675 | 0.673 | 0.675  |
| DT     | 0.487| 0.599 | 0.606 | 0.628 | 0.620  |
| GB     | 0.528| 0.597 | 0.630 | 0.626 | 0.624  |
| LSVC   | 0.495| 0.630 | 0.650 | 0.657 | 0.654  |
| SVC    | 0.492| 0.577 | 0.587 | 0.606 | 0.589  |
| LOG    | 0.445| 0.600 | 0.632 | 0.648 | 0.650  |
| **code-stat**                                  |
| RF     | 0.666| **0.716**| **0.725**| **0.724**| **0.705**|
| DT     | 0.612| 0.647 | 0.668 | 0.672 | 0.681  |
| GB     | 0.619| 0.674 | 0.671 | 0.668 | 0.675  |
| LSVC   | 0.506| 0.638 | 0.646 | 0.655 | 0.657  |
| SVC    | 0.496| 0.572 | 0.590 | 0.596 | 0.588  |
| LOG    | 0.458| 0.616 | 0.643 | 0.655 | 0.655  |
| **code-cell**                                  |
| RF     | 0.601| 0.640 | 0.632 | 0.634 | 0.634  |
| DT     | 0.470| 0.532 | 0.545 | 0.563 | 0.584  |
| GB     | 0.541| 0.617 | 0.616 | 0.623 | 0.636  |
| LSVC   | 0.464| 0.583 | 0.577 | 0.576 | 0.581  |
| SVC    | 0.435| 0.458 | 0.449 | 0.420 | 0.473  |
| LOG    | 0.456| 0.565 | 0.574 | 0.597 | 0.617  |
| **all-features**                              |
| RF     | **0.668**| 0.672 | 0.674 | 0.670 | 0.663  |
| DT     | 0.537| 0.561 | 0.603 | 0.610 | 0.627  |
| GB     | 0.655| 0.680 | 0.671 | 0.667 | 0.673  |
| LSVC   | 0.475| 0.590 | 0.586 | 0.579 | 0.590  |
| SVC    | 0.433| 0.471 | 0.478 | 0.426 | 0.480  |
| LOG    | 0.460| 0.574 | 0.585 | 0.595 | 0.617  |
| **all-features-without-code**                  |
| RF     | 0.567| 0.590 | 0.588 | 0.584 | 0.574  |
| DT     | 0.462| 0.438 | 0.438 | 0.445 | 0.457  |
| GB     | 0.493| 0.579 | 0.591 | 0.561 | 0.550  |
| LSVC   | 0.353| 0.447 | 0.459 | 0.478 | 0.475  |
| SVC    | 0.325| 0.353 | 0.344 | 0.293 | 0.341  |
| LOG    | 0.346| 0.419 | 0.431 | 0.452 | 0.458  |

Bold numbers indicate column-wise top performers and numbers in coloured background cells indicate row-wise top performers.

Given a current state in the transition, we compare the probabilities to the next state between the predicted set and DASWOW using Kendall’s tau. Hence, we essentially compare each row of Fig. 5 with the respective row of Fig. 12 to compute the correlation. Table 12 shows the correlation results. We found a statistically significant correlation ($p <= 0.05$) between the state transition probabilities in these figures, which shows the
effectiveness of the classifier in predicting the labels resulting in comparable transition sequences.

5.3.1 Summary

Our workflow analysis (Section 5.3) on the predictions from classifier reveals similar insights to Section 4 on individual data science steps. We believe that the performance of the classifier can be enhanced by other approaches, including interactive methods like active learning leading to a more accurate data science workflow analysis. Such an analysis would

Fig. 11 Number of cells per label: all labels

Fig. 12 Transition matrix probabilities across data science workflow - predicted
Table 12  Kendall’s correlation coefficient of transition probabilities to the next state between the training and predicted set given a current state in the data science workflow

| current_state          | Kendall’s tau |
|------------------------|---------------|
| start                  | 0.861         |
| helper_functions       | 0.881         |
| load_data              | 0.697         |
| data_exploration       | 0.891         |
| data_preprocessing     | 0.709         |
| modelling              | 0.673         |
| evaluation             | 0.709         |
| prediction             | 0.673         |
| result_visualization   | 0.661         |
| save_results           | 0.611         |
| comment_only           | 0.740         |

reveal patterns and provide insights that support data science development and automation of data science workflows.

6 Discussion and Implications

In this section, we further discuss the insights revealed through our workflow analysis on data science code using DASWOW and highlight its implications along two dimensions: the development and maintenance of data science workflows. We also identify and categorise the use cases where the data science step annotations can be of value.

Better tool support for computational notebook IDEs  Data scientists lack proper coding assistance within computational notebooks (Chattopadhyay et al. 2020; Olabarriaga et al. 2014). Features that already exist in traditional IDEs should be extended to computational notebooks like Jupyter as “data scientists are akin to software developers, but constrained to a very specialised environment” (Olabarriaga et al. 2014). Here, data science annotations can act as task tags or provide metadata information in IDE tools. For example, they can support type hierarchy (Robillard et al. 2004) by grouping data science procedures since data scientists currently cannot “explore the API and functions” in the notebook (Chattopadhyay et al. 2020). They can help data scientists in (re-)structuring a notebook, one of the pain points in notebook development (Head et al. 2019; Titov et al. 2022; Dong et al. 2021), by identifying the activities they perform and splitting and(or) merging the cells.

Supporting exploratory programming  Data scientists spend a large fraction of cells (as a result, time and effort) exploring and pre-processing data (refer to Section 4.1). While the effort required for pre-processing tasks has already received attention for automation support, the exploration itself has been traditionally considered a manual task left to the data scientists and domain experts. There is still room for creating tools that automate data exploration in order to assist data scientists. For example, a tool that provides a sample set of exploration paths (“garden of forking paths”) (Gelman and Loken 2013; Schweinsberg et al. 2021) to be tested by the user would be useful.
At times, data scientists implement more than one data science step per cell (refer to Section 4.1). Our results provide additional empirical evidence to the finding that notebooks are usually ‘messy’ and lack logical structuring (Head et al. 2019; Kery et al. 2019; Rule et al. 2018) due to the exploratory nature of data science coding. With the help of data science annotation, tools could support the automatic modularisation of a code block by combining or merging cells within the notebook.

Furthermore, data scientists transition within and between data science steps iteratively throughout a data science workflow (refer to Section 4.2). Together with a lack of structure, this can lead to navigation and comprehension difficulties for data scientists trying to understand others’ or their own notebooks and further hinder collaboration (Head et al. 2019; Kery et al. 2019; Chattopadhyay et al. 2020). We also observe in the transition patterns that, many times, a cell performing a certain activity is followed by another cell that performs the same activity. This behaviour is not intuitive. For instance, a cell performing data exploration activity is followed by another cell which performs the same activity 43% of the time. This finding calls for larger and more specific support to data scientists in the context of workflow design, development, and management. However, to the best of our knowledge, current workflow automation systems do not take into account this iterative pattern of steps in a workflow (Heffetz et al. 2020; Drori et al. 2021).

**Automating data science coding** Chattopadhyay et al. (2020) identify lack of support for coding (e.g. live templates, programming examples) as one of the pain points for data scientists. As a result, data scientists spend a lot of time browsing examples on the internet (e.g., Stack Overflow) during their development (Chattopadhyay et al. 2020). By integrating a tool or a plugin that dynamically classifies the code being written according to the data science step, a programming environment can suggest relevant coding templates. Particularly, such a support in modelling and data exploration steps may be helpful as they are frequently cloned (refer to Section 4.4) and may cause code dependency problems (Kery et al. 2018) in the notebook. Such a tool may also learn from the user’s acceptance of suggested annotations to improve itself and customise its future recommendations. They can also provide complementary support to existing recommender systems (Watson et al. 2019; Svyatkovskiy et al. 2019) (that focus on certain steps) by automatically identifying and predicting the current (or next) data science activity so that relevant examples and alternatives (e.g., a template recommender tool may use the identified step in order to provide appropriate examples) are suggested to the data scientist.

Our workflow analysis also reveals that data scientists invoke or import many helper functions, including external libraries, while performing a data science task. Identifying the right libraries to use and managing their versions (Wang et al. 2021b; Chattopadhyay et al. 2020) can become time-consuming and inconvenient for data scientists over time. By predicting the next data science step, an automatic library recommender and manager with an in-built function of adding the code can then recommend relevant libraries.

**Support in navigation and comprehension of data science notebooks** Our empirical findings show several potential issues that may give rise to navigation difficulties, as also reported by other interview-based studies (Chattopadhyay et al. 2020; Head et al. 2019; Kery et al. 2019). Implementing multiple steps that require logical separation within the same cell (Head et al. 2019), multiple iterations (Chattopadhyay et al. 2020), lack of an overview of the workflow — data science steps and the transitions between them due to the notebook’s linear structure, could all contribute to a messy notebook that is difficult to navigate through and comprehend. In such cases, data science step annotations can act similar
to pre-defined annotations in traditional IDEs. Together with a code browser, they can allow the users to navigate, access, and debug the relevant block of code easily (Storey et al. 2009; McCormick and De Volder 2004).

**Support in workflow management** Data science annotations can support transformation and integration of data science notebooks as workflows into other scientific workflow systems like Taverna and Wings (Garijo et al. 2013a). Furthermore, they can enable the analysis of the evolution of the data science workflows and enable users to find, interpret, modify, and reuse relevant (sub-)workflows.

7 Limitations and Threats to Validity

One limitation in our study is that our annotation process during the dataset generation relied on few annotators. However, we selected the annotators after a pilot study in order to generate a dataset with consistent and high-quality annotations. Further, given expert annotations are expensive and hard to obtain, improvements to the dataset can be introduced in future using techniques like active learning. In addition to the disagreements we discussed in Section 3.3, our annotation task also revealed the difficulties that arise in understanding a notebook (Rule et al. 2018; Pimentel et al. 2019). For example, a cell may contain a function definition that is executed only later in the notebook. In such cases, we refer to other steps that may be present in the cell as a primary task instead of the function definition. Another example of ambiguity arises if a cell contains code that performs several activities. Here, deciding a single *primary* label is difficult. While annotating each line of code may be a solution, this is, however, tedious for annotators and also expensive. We address this currently by preferring the first appearing step as *primary* label and marking the others as *secondary* labels. We see this as a good compromise given their low frequency of appearance (refer to Table 6).

Another limitation is that the supervised classification method does not perform well on certain labels. For example, *evaluation*, *prediction*, and *result* visualization labels had a comparatively low F1-score. This is due to the unbalanced nature of the dataset, i.e., the rare appearance of these labels in the dataset. From our analysis, we view this as an inherent disadvantage that should be tackled with upampling. In the case of *result* visualization, we suspect another reason for the low classifier performance is that the code structure is similar to *data* preprocessing and *data* exploration. We believe this could be improved by considering the sequential information around the label and is left for future work.

Validity concerns arise primarily from generalising the results of a set of notebooks that are taken from GitHub. To mitigate this issue, we applied random sampling to select the notebooks from the widely used corpus by Rule et al. (2018) to create DASWOW. Also, the descriptive statistics on DASWOW corroborates with findings on the entire corpus. However, we cannot rule out that the results from workflow analysis of notebooks from other platforms like Kaggle, where data scientists compete with peers and write well-documented notebooks (Gil et al. 2010), may diverge.

Another possible concern could be the representativeness of the sample analysed in this work. To address this issue and also to enable further validation, we relied on the widely used notebooks dataset created by Rule et al. (2018) as the source. We selected the notebooks through random sampling and through then manually verified the purpose of each of the notebooks before considering them for our analyses.
8 Conclusion

Understanding the way data science is designed, implemented and reported by data scientists is important to provide tools and frameworks that enable high-quality data science. In this paper, we presented an empirical analysis of the data science workflows in real-world practical implementations. To that end, we first provide an expert-annotated dataset (DASWOW) for workflow analyses. Our analysis confirms previous preliminary evidence that most real-world notebooks have iterative nature and reveal unique patterns in the data science workflow.

Through an evaluation of existing classification methods, we also showcased that DASWOW can help in extending the dataset via machine learning methods. To that end, we evaluated a set of classifiers for several sets of features in order to understand their relevance in the classifier performance. We also extended our workflow analysis on the predicted set of labels. Our results indicate that such an extension via machine learning methods is indeed possible and provide a baseline for future improvements (e.g., via more sophisticated feature construction or deep learning techniques).

The results of our workflow analysis also provide valuable insights and motivation for developing tools that support, guide, and automate data science coding. Among other things, they raise questions about how to support the clearly iterative nature of data science explorations and the need for further understanding of the intermingling of various tasks within cells. As such, we hope that data science and the insights provided by our analyses will help (1) build better tools for supporting data scientists’ analyses and (2) bootstrap further analyses of other data science notebook collections.

Appendix A

A.1 Instructions for Ground Truth Annotators

The main goal of this task is to design and implement a method that automatically classifies different parts of a Data Science notebook, so as to label the steps of the Data Science process that are present in the notebook. In order to train and evaluate the method, we need to curate a ground truth. Therefore, your input as an expert labeller is very much appreciated.

You will be given a set of notebooks to be labelled, a set of possible labels and your task is to indicate for each cell of the notebook one or more relevant label(s), that describe the purpose of the code appearing in the cell. We would also like you to indicate how confident you feel about your answers. Below, you will find detailed information about the meaning of the labels and things to take into account.

In this annotation task, you will need to consider the following files for each notebook to be annotated:

- **notebook_name.ipynb**: the Python notebook to be analysed and labelled.
- **notebook_name.ipynb.xlsm**: a macro-enabled excel worksheet, where you should annotate the notebook.
A.1.1 Instructions

Please read the instructions before proceeding with the labelling task.

How to annotate one notebook

- Open the Python notebook and the annotation file in any compatible software (e.g., use Jupyter for the notebook and Excel for the annotation file).
- Read the set of possible labels and their meaning (see also Section Appendix).
- For each cell in the notebook, please read the content and consider the line(s) of code, plots, visualizations and comments. Please ignore markdown.
- Decide the purpose of the cell, and assign one or more of the available labels. In the annotation file (notebook_name.ipynb.xlsm), please mark with 1 the label(s) you would like to assign to the cell.
- Also, determine the primary data science process performed in the cell. Record the observation by choosing the appropriate label in the primary_label column in the annotation file.
- If, for a given cell, you do not agree with any of the given labels, please indicate the label that you think is relevant in the other_label column.
- You have the option of adding some notes in the notes column, if you would like to enter any observation about the cell (e.g., if it was hard to decide the label or anything you think it is important to be mentioned). You do not need to enter notes for every single cell.
- Once you have completed the labelling of all cells, please indicate in the last two rows of annotation file the following information:
  1. confidence percentage: how confident you felt when providing the labels (with a value between 0 (not confident at all) and 100 (very confident)).
  2. notebook score: give a general score for the notebook, based on the clarity and understandability of the code (with a value between 0 (not clear at all) and 100 (very clear)).

General tips

- The annotation should be done based on the available implementation in the notebook. Please do not execute the code of the notebook to decide the label.
- Note that sometimes, it is useful to go through the whole notebook to be aware of the context of the cells.

A.1.2 Classification labels

We defined a set of 9 labels: 3 of them are labels that refer to general coding actions, and 7 labels correspond to the main steps in the data science process.

General labels

load_data This label indicates that the cell is just loading data into the Jupyter notebook environment. The code can be for loading one or multiple data sets of any type (e.g., .csv, .pkl, .jpg, .png, .hdf5).
helper_functions This label should be used when the cell contains code that supports the execution of the code in the notebook. These helper functions can be *import statements* (e.g., `import pandas as pd`) or other piece of code that is not directly related to the data science activity at hand and rather are useful functions in scripting. For example, built-in Jupyter notebook magic commands. A concrete example would be: `%matplotlib inline`, which sets the inline backend so that the output plots are displayed directly below the code cell that executes it. Some other examples include `from IPython.display import import Audio, %pprint`.

comment_only If the cell contains only commented text, you should use this label. This label should not be used for markdown; please remember that you should ignore markdown.

Labels for each of the data science steps

data_preprocessing Data preprocessing includes tasks such as cleaning, instance selection, data normalization, data transformation, feature extraction and feature selection. Data preprocessing ensures that the data does not contain irrelevant, redundant and inconsistent data.20

data_exploration Data exploration21 or exploratory data analysis is an approach to initial data analysis, where a data scientist inspects the content of a dataset in order to understand the nature and characteristics of the data. This step, that often contributes to the identification of patterns in the data, is helpful to identify research hypotheses and decide on the modelling. Data exploration often involves visual exploration of the data.

modelling Modelling is the process of applying (or fitting) statistical models and algorithms to the data in order to “perform a specific task without explicit restrictions, relying on models and inferences instead”22. In simple terms, training a classifier or a neural network to learn from the data is modelling.

prediction Prediction is an important step as many of the data science tasks are predictive modelling tasks. This label refers to the generation of an outcome that was previously unknown, by applying the model built to new data.

evaluation Evaluation refers to the process of evaluating the model using evaluation metrics like accuracy, or F1-score. In the context of model selection, the evaluation step can be used to compare different models. Evaluation is also done to compare different models23.

result_visualization Result visualization is the graphical representation of results (via elements like plots, tables and other graphs). Please note that we differentiate between data exploration and result visualization. The purpose of the former is to better understand the data, while the latter focuses on visualizing e.g. a tested hypothesis, or a performance comparison.

---

20Data preprocessing https://en.wikipedia.org/wiki/Data_pre-processing
21Data exploration https://en.wikipedia.org/wiki/Data_exploration
22https://en.wikipedia.org/wiki/Machine_learning#Models
23Evaluating Machine Learning Models https://www.oreilly.com/data/free/evaluating-machine-learning-models.csp for more information
| Feature | Description | Rationale for inclusion |
|---------|-------------|------------------------|
| **Cell features** are extracted from the content of individual cells, their related components (such as the output obtained from the cell execution), and information about neighbouring cells. | **filename**† | Filename indicates the name of the notebook file where the cell comes from. It provides metadata information about the cell by providing information about where it resides. Cells containing the same filename but with a different cell number provide positional information about the cell that may indicate the presence of certain data science steps. |
| | **cell_number** | Cell number indicates the position of the cell in a given notebook. Cell number along with filename uniquely identifies a cell in the dataset. This information about a code cell may be indicative of the type of the data science action in the cell as different data science steps are expected to occur at a certain phase in the cycle (Wang et al. 2019a; Zhang et al. 2020a; Aggarwal et al. 2019; Muller et al. 2019b; Souza et al. 2020). For e.g., data preparation and data exploration generally occurs in the initial stages of data science workflow whereas modelling and evaluation at a later stage. |
| | **execution_count** | Execution count indicates the order in which a cell was executed by the user in a notebook. It provides execution-level information about a code cell which could be indicative of a certain data science step. For example, a lower number may suggest a step at the beginning of the data science workflow. |
| | **text** | Text contains the content of the cell, which includes either the code text or markdown text depending on the cell_type (markdown or actual executable code). Different blocks of code implement different data science actions and hence can exhibit patterns that identify the data science step. |
| | **comment** | All the comments in a given code cell. This feature is valid only for code cells. It provides (additional) information about the intended purpose of the code present in a particular cell (Rule et al. 2018). |
| | **output_text** | This indicates the content of the output. For example, image/png or text/plain. This feature is valid for the cells of cell_type output. The output information of a given code cell may be indicative of a certain data science step. For example, image/png may indicate a data exploration step since data scientists use visualisation techniques to explore data. |
| | **output_type** | This indicates the type of the output. For example, display_data. This feature is valid for the cells of cell_type output. It provides information about the type of the output (stream data or rich mime-type output) that could be indicative of a certain data science step. |
| | **code_line_before** | code_line_before indicates the last line of code in a code cell preceding the current cell. In the case of the cells of cell_type markdown or raw_nb, this information is obtained from the code cell preceding it (may or may not be immediately preceding). The previous code cell’s line of code provides additional context by providing information regarding the surrounding structure of a particular data science step (Bacchelli et al. 2012). |
| Feature               | Description                                                                 | Rationale for inclusion                                                                 |
|----------------------|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| code_lines_after     | code_lines_after indicates the first line of code in a code cell following  | The following code cell’s line of code provides additional contextual                   |
|                      | the current cell. In the case of the cells of cell_type markdown or raw_nb,  | information and may provide indications about the previous step.                        |
|                      | this information is obtained from the code cell following the current cell   |                                                                                        |
|                      | (may or may not be immediately succeeding).                                 |                                                                                        |
| markdown_heading     | markdown_heading indicates the first line of a markdown cell.                | They are descriptive of the steps in the code cells that follow because data            |
|                      | In the case of the cells of cell_type code, this information is             | scientists tend to organise their code in sections for readability purposes            |
|                      | obtained from the markdown cell preceding it (may or may not be immediately | ((Rule et al. 2018; Bacchelli et al. 2012)).                                            |
|                      | preceding).                                                                 |                                                                                        |
| External features    | containing the detailed textual description of the imported libraries into  |                                                                                        |
| packages_info        | This feature indicates the man-page information about the libraries imported | It provides information about the libraries imported in the cell                       |
|                      | in it provides information about the libraries imported in the cell         | a given cell. The information contains the description of libraries as given in an      |
|                      | only cell. The information contains the description of libraries as given in | using a detailed description, thus providing a larger bag of                          |
|                      | an using a detailed description, thus providing a larger bag of              | distinctive words.                                                                      |
|                      | external source: The Python Package Index (pypi)³                         |                                                                                        |
| Statistical/Metric features | Statistical or Software Metric-based features are features that contain code | It provides statistical information about the density of a code cell which could be       |
|                      | metrics for a given cell. Below custom metrics are                         | indicative of the data science step because certain activities require a lot of lines of |
|                      | generated using class:NotebookMetrics and class:CodeCellMetrics. The       | code than others. For example, data_preprocessing may have a lot of lines of code.      |
|                      | metric-based features are included under the view that structural           |                                                                                        |
|                      | information of the notebook can provide information about the data science  |                                                                                        |
| no_lines_of_code     | Lines of code indicates the total number of lines in a given cell of cell_  |                                                                                       |
|                      | type code.                                                                   |                                                                                        |
| no_lines_of_comment  | Lines of comment indicates the total number of comment lines in a given     |                                                                                       |
|                      | cell_of cell_type code. Any line that starts with ‘#’ is considered          |                                                                                       |
|                      | a comment line.                                                             |                                                                                        |
| function_count       | Function count indicates the number of functions in a given cell of cell_    |                                                                                       |
|                      | type code. Any line that starts with the keyword ‘def’ is considered         |                                                                                       |
|                      | to be a function.                                                           |                                                                                        |
| variable_count       | Variable count indicates the number of variables in a given cell of cell_   |                                                                                       |
|                      | type code. A variable is considered to be any keyword composed of             |                                                                                       |
|                      | alphanumerics and _ with a ‘=’ to the right. Parameters are not              |                                                                                       |
|                      | considered.                                                                 |                                                                                        |

³https://pypi.org/
**save_results** This label should be used when the cell is primarily creating a persistent copy of some results (e.g. serializing the content of a variable into a CSV file).

### A.2 List of features

Table 13 lists all the features\(^{24}\) extracted to form the DASWOW.

### Funding

Open access funding provided by University of Zurich. This research work was partly funded by the Swiss National Science Foundation (SNSF) under contract numbers 200020\_184994, 407540\_167303 and PP00P2\_170529.

### Data Availability

The datasets generated and analysed during the current study are available as a part of our replication package at the following link: https://doi.org/10.5281/zenodo.5635475

### Code Availability

All data and materials that support the findings of this study are available in our replication package at the following link: https://doi.org/10.5281/zenodo.5635475

### Declarations

**Competing interests** The authors declare they have no financial or non-financial interests.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

### References

Aggarwal C, Bouneffouf D, Samulowitz H, Buesser B, Hoang T, Khurana U, Liu S, Pedapati T, Ram P, Rawat A, Wistuba M, Gray A (2019) How can ai automate end-to-end data science?arXiv:1910.14436

Altman DG (1990) Practical statistics for medical research. CRC press, Florida

Aragon C, Hutto C, Echenique A, Fiore-Gartland B, Huang Y, Kim J, Neff G, Xing W, Bayer J (2016) Developing a research agenda for human-centered data science. In: Proceedings of the 19th ACM conference on computer supported cooperative work and social computing companion, pp 529–535

Bacchelli A, Dal Sasso T, D’Ambros M, Lanza M (2012) Content classification of development emails

Barstad V, Goodwin M, Gjeseter T (2014) Predicting source code quality with static analysis and machine learning. In: Norsk IKT-konferanse for forskning og utdanning

Bennett KP, Erickson JS, de Los Santos H, Norris S, Patton E, Sheehan J, McGuinness DL (2016) Data analytics as data: a semantic workflow approach. In: Proc of artificial intelligence for data science workshop at neural information processing systems (NIPS), Barcelona, Spain

Berthold MR, Cebron N, Dill F, Gabriel TR, Köttter T, Meinel T, Ohl P, Thiel K, Wiswedel B (2009) Knime: the konstanz information miner: version 2.0 and beyond. ACM SIGKDD Explor Newsl 11(1):26–31

Brown WH, Malveau RC, McCormick HWS, Mowbray TJ (1998) AntiPatterns: refactoring software, architectures and projects in crisis. Wiley, New Jersey

Carvalho LA, Wang R, Gil Y, Garijo D (2017) Niw: converting notebooks into workflows to capture dataflow and provenance. In: K-CAP workshops, pp 12–16

Carvalho LAM, Garijo D, Medeiros CB, Gil Y (2018) Semantic software metadata for workflow exploration and evolution. In: 2018 IEEE 14th International Conference on e-Science (e-Science), IEEE, pp 431–441

Chan DK, Leung KR (1997) A workflow vista of the software process. In: Database and expert systems applications. 8th international conference, DEXA’97 Proceedings, IEEE, pp 62–67

\(^{24}\)† same for all the code cells (data points) for a given notebook.
Chattopadhyay S, Prasad I, Henley AZ, Sarma A, Barik T (2020) What’s wrong with computational notebooks? pain points, needs, and design opportunities. In: Proceedings of the 2020 CHI conference on human factors in computing systems, pp 1–12

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20:37–46

Colombo A, Damiani E, Gianini G (2006) Discovering the software process by means of stochastic workflow analysis. J Syst Archit 52(11):684–692

Desmond Y (2020) Structuring jupyter notebooks for fast and iterative machine learning experiments. https://towardsdatascience.com/, Accessed on 01 Jan 2021

Dong H, Zhou S, Guo JL, Kastner C (2021) Splitting, renaming, removing: a study of common cleaning activities in jupyter notebooks. In: 2021 36th IEEE/ACM international conference on automated software engineering workshops (ASEW), pp 114–119. https://doi.org/10.1109/ASEW52652.2021.00032

Drori I, Krishnamurthy Y, Rampin R, Lourenco RdP, Ono JP, Cho K, Silva C, Freire J (2021) Alphad3m: machine learning pipeline synthesis. arXiv:211102508

Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston

Garrio D, Alper P, Belhajjame K, Corcho O, Gil Y, Goble C (2013a) Common motifs in scientific workflows: an empirical analysis. Future Gener Comput Syst 36:338–351. https://doi.org/10.1016/j.future.2013.09.018

Garrio D, Corcho O, Gil Y (2013b) Detecting common scientific workflow fragments using templates and execution provenance. In: Proceedings of the seventh international conference on Knowledge capture, pp 33–40

Gelman A, Loken E (2013) The garden of forking paths: why multiple comparisons can be a problem, even when there is “fishing expedition” or “p-hacking” and the research hypothesis was posited ahead of time. Dep Stat Columbia Univ 348

Gil Y, Ratnakar V, Kim J, Gonzalez-Calero P, Groth P, Moody J, Deelmaen E (2010) Wings: intelligent workflow-based design of computational experiments. IEEE Intell Syst 26(1):62–72

Guo PJ, Seltzer M (2012) Burrito: wrapping your lab notebook in computational infrastructure. In: Proceedings of the 4th USENIX conference on theory and practice of provenance, TaPP’12. USENIX Association, USA, p 7

Head A, Hohman F, Barik T, Drucker SM, DeLine R (2019) Managing messes in computational notebooks. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI ’19. Association for Computing Machinery, New York, pp 1–12. https://doi.org/10.1145/3290605.3300500

Hefetz Y, Vainshtein R, Katz G, Rokach L (2020) Deepline: automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering. In: Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining, pp 2103–2113

Hernández-Orallo J, Vold K (2019) Ai extenders: The ethical and societal implications of humans cognitively extended by ai. In: Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society, pp 507–513

Hofmann M, Klinkenberg R (2016) RapidMiner: data mining use cases and business analytics applications. CRC Press, Florida

Hohman F, Wongsuphasawat K, Kery MB, Patel K (2020) Understanding and visualizing data iteration in machine learning. In: Proceedings of the 2020 CHI conference on human factors in computing systems, pp 1–13

Jupyter P (2015) Project jupyter: computational narratives as the engine of collaborative data science. https://jupyter.org/

Källén M, Wrigstad T (2020) Jupyter notebooks on github: characteristics and code clones. arXiv:200710146

Kandell S, Paepcke A, Hellerstein JM, Heer J (2012a) Enterprise data analysis and visualization: an interview study. IEEE Trans Vis Comput Graph 18(12):2917–2926. https://doi.org/10.1109/TVCG.2012.219

Kandell S, Paepcke A, Hellerstein JM, Heer J (2012b) Enterprise data analysis and visualization: an interview study. IEEE Trans Vis Comput Graph 18(12):2917–2926

Keith B, Vega V (2016) Process mining applications in software engineering. In: International conference on software process improvement, Springer, pp 47–56

Kery MB, Horvath A, Myers B (2017) Variolite: supporting exploratory programming by data scientists. In: Proceedings of the 2017 CHI conference on human factors in computing systems, CHI ’17. Association for Computing Machinery, New York, pp 1265–1276. https://doi.org/10.1145/3025453.3025626

Kery MB, Radensky M, Arya M, John BE, Myers BA (2018) 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, CHI ’18. Association for Computing Machinery, New York, pp 1–11. https://doi.org/10.1145/3173574.3173748

Kery MB, John BE, O’Flaherty P, Horvath A, Myers BA (2019) Towards effective foraging by data scientists to find past analysis choices. In: Proceedings of the 2019 CHI conference on human
factors in computing systems, CHI ’19. Association for Computing Machinery, New York, pp 1–13. https://doi.org/10.1145/3290605.3300322
Kim M, Zimmermann T, DeLine R, Begel A (2016) The emerging role of data scientists on software development teams. In: Proceedings of the 38th international conference on software engineering, ICSE ’16. Association for Computing Machinery, New York, pp 96–107. https://doi.org/10.1145/2884781.2884783
Knab P, Pinzger M, Bernstein A (2006) Predicting defect densities in source code files with decision tree learners. In: Proceedings of the 2006 international workshop on Mining software repositories, pp 119–125
Koenzen AP, Ernst NA, Storey MAD (2020) Code duplication and reuse in jupyter notebooks. In: 2020 IEEE symposium on visual languages and human-centric computing (VL/HCC), pp 1–9. https://doi.org/10.1109/VL/HCC50065.2020.9127202
Krämer JP, Karrer T, Kurz J, Wittenhagen M, Borchers J (2013) How tools in ide shape developers’ navigation behavior. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp 3073–3082
Kross S, Guo PJ (2019) Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In: Proceedings of the 2019 CHI conference on human factors in computing systems, CHI ’19. ACM, New York, pp 263:1–263:14. https://doi.org/10.1145/3290605.3300493
Kun P, Mulder I, Kortuem G (2018) Design enquiry through data: appropriating a data science workflow for the design process. In: Proceedings of the 32nd international BCS human computer interaction conference, vol 32, pp 1–12
LaToza TD, Myers BA (2010) Hard-to-answer questions about code. In: Evaluation and usability of programming languages and tools, pp 1–6
Lee A, Xin D, Lee D, Parameswaran A (2020) Demystifying a dark art: understanding real-world machine learning model development. arXiv:200501520
Lever J, Krzywinski M, Altman NS (2016) Points of significance: Model selection and overfitting. Nat Methods 13:703–704
Liu J, Pacitti E, Valduriez P, Mattoso M (2015) A survey of data-intensive scientific workflow management. J Grid Comput 13(4):457–493
Liu J, Boukhelfa N, Eagan JR (2020) Understanding the role of alternatives in data analysis practices. IEEE Trans Vis Comput Graph 26:66–76
Mao Y, Wang D, Muller MJ, Varshney KR, Baldini I, Dugan C, Mojsilovic A (2019) How data scientists work together with domain experts in scientific collaborations. Proc ACM Human-Comput Interact 3:1–23
McCormick E, De Volder K (2004) Jquery: finding your way through tangled code. In: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp 9–10
Meena HK, Saha I, Mondal KK, Prabhakar T (2005) An approach to workflow modeling and analysis. In: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, pp 85–89
Menzies T, Williams L, Zimmermann T (2016) Perspectives on data science for software engineering. Morgan Kaufmann, Burlington
Microsoft (2020) What is the team data science process?. https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview, Accessed 1 Jan 2021
Miller S, Hughes D (2017) The quant crunch. How the demand for data science skills is disrupting the job market. Burning Glass Technologies, Boston
Missier P, Soiland-Reyes S, Owen S, Tan W, Nenadic A, Dunlop I, Williams A, Oinn T, Goble C (2010) Taverna, reloaded. In: International conference on scientific and statistical database management, Springer, pp 471–481
Muir BM (1994) Trust in automation: part i. theoretical issues in the study of trust and human intervention in automated systems. Ergonomics 37(11):1905–1922
Muller M, Feinberg M, George T, Jackson SJ, John BE, Kery MB, Passi S (2019a) Human-centered study of data science work practices. In: Extended abstracts of the 2019 CHI conference on human factors in computing systems, pp 1–8
Muller M, Lange I, Wang D, Piorkowski D, Tsay J, Liao QV, Dugan C, Erickson T (2019b) How data science workers work with data: discovery, capture, curation, design, creation. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI ’19. Association for Computing Machinery, New York, pp 1–15. https://doi.org/10.1145/3290605.3300356
Neill CJ, Laplante PA, DeFranco JF (2011) Antipatterns: managing software organizations and people. CRC Press, Florida
Norman DA (1990) The ‘problem’ with automation: inappropriate feedback and interaction, not ‘over-automation’. Philos Trans R Soc Lond B Biol Sci 327(1241):585–593
Olabarriaga S, Pierantoni G, Taffoni G, Sciacca E, Jaghoori M, Korkhov V, Castelli G, Vuerli C, Becciani U, Carley E, et al. (2014) Scientific workflow management—for whom? In: 2014 IEEE 10th international conference on e-science, vol 1. IEEE, pp 298-305

O’Neil C, Schutt R (2013) Doing data science. Straight talk from the frontline. O’Reilly Media Inc., California

Parasuraman R, Sheridan TB, Wickens CD (2000) A model for types and levels of human interaction with automation. IEEE Trans Syst Man Cybernet Part A Syst Hum 30(3):286–297

Park LA, Read J (2018) A blended metric for multi-label optimisation and evaluation. In: Joint european conference on machine learning and knowledge discovery in databases, Springer, pp 719–734

Pascarella L, Bacchelli A (2017) Classifying code comments in java open-source software systems. In: 2017 IEEE/ACM 14th international conference on mining software repositories, MSR, IEEE, pp 227–237

Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. J Syst Softw 150:22–36

Passi S, Jackson SJ (2018) Trust in data science: collaboration, translation, and accountability in corporate data science projects. Proc ACM Human-Comput Interact 2(CSCW):1–28

Patterson E, McBurney R, Schmidt H, Baldini I, Mojsilović A, Varshney KR (2017) Dataflow representation of data analyses: toward a platform for collaborative data science. IBM J Res Dev 61(6):9–1

Pellin BN (2000) Using classification techniques to determine source code authorship. White Paper, Department of Computer Science, University of Wisconsin

Pimentel JaF, Murta L, Braganholo V, Freire J (2019) A large-scale study about quality and reproducibility of jupyter notebooks. In: Proceedings of the 16th international conference on mining software repositories, IEEE Press, MSR ’19, p 507–517. https://doi.org/10.1109/MSR.2019.00077

PriceWaterhouseCoopers (2017) Investing in america’s data science and analytics talent: a case for action. In: Business-higher education forum report

Rahman F, Bird C, Devanbu P (2012) Clones: what is that smell? Empir Softw Eng 17(4):503–530

Robillard MP, Coelho W, Murphy GC (2004) How effective developers investigate source code: An exploratory study. IEEE Trans Softw Eng 30(12):899–903

Roy CK, Cordy JR (2007) A survey on software clone detection research. Queen’s School Comput TR 54(115):64–68

Rubin V, Güntner CW, Van Der Aalst WM, Kindler E, Van Dongen BF, Schäfer W (2007) Process mining framework for software processes. In: International conference on software process, Springer, pp 169–181

Rule A, Tabard A, Hollan JD (2018) Exploration and explanation in computational notebooks. In: Proceedings of the 2018 CHI conference on human factors in computing systems, CHI ’18. Association for Computing Machinery, New York, pp 1–2. https://doi.org/10.1145/3173574.3173606

Schweinsberg M, Feldman M, Staub N, van den Akker OR, van Aert RC, Van Assen MA, Liu Y, Althoff T, Heer J, Kale A, et al. (2021) Same data, different conclusions: Radical dispersion in empirical results when independent analysts operationalize and test the same hypothesis. Organizational Behavior and Human Decision Processes

Smith B, Mizell D, Gilbert J, Shah V (2005) Towards a timed markov process model of software development. In: Proceedings of the second international workshop on Software engineering for high performance computing system applications, pp 65–67

Souza R, Azevedo LG, Lourenço V, Soares E, Thiago R, Brandão R, Civitarese D, Brazil EV, Moreno M, Valduriez P, Mattoso M, Cerqueira R, Netto MAS (2020) Workflow provenance in the lifecycle of scientific machine learning

Springboard (2016) The data science process. https://www.kdnuggets.com/2016/03/data-science-process.html, Accessed 1 Jan 2021

Storey MA, Ryall J, Singer J, Myers D, Cheng LT, Muller M (2009) How software developers use tagging to support reminding and refinding. IEEE Trans Softw Eng 35(4):470–483

Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N (2019) Pythia: ai-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pp 2727–2735

Titov S, Golubev Y, Bryksin T (2022) Resplit: improving the structure of jupyter notebooks by re-splitting their cells. In: 2022 IEEE international conference on software analysis, evolution and reengineering (SANER), pp 492–496. https://doi.org/10.1109/SANER53432.2022.00066

Trcka N, Aalst V, Sidorova N (2008) Analyzing control-flow and data-flow in workflow processes in a unified way. Computer science report

Tsoumakas G, Vlahavas I (2007) Random k-labelsets: an ensemble method for multilabel classification. In: ECML.
UCSD C (2021) Introduction to big data - steps in the data science process. coursera (university of california san diego). https://www.coursera.org/lecture/big-data-introduction/steps-in-the-data-science-process-Fonq2, Accessed 1 Jan 2021

Ugurel S, Krovetz R, Giles CL (2002) What’s the code? automatic classification of source code archives. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp 632–638

Vassiliadis P, Simitsis A, Baikousi E (2009) A taxonomy of etl activities. In: Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP, pp 25–32

Wang D, Weisz JD, Muller M, Ram P, Geyer W, Dugan C, Tausczik Y, Samulowitz H, Gray A (2019a) Human-ai collaboration in data science. In: Proceedings of the ACM on human-computer interaction 3(CSCW):1–24. https://doi.org/10.1145/3359313

Wang D, Liao QV, Zhang Y, Khurana U, Samulowitz H, Park S, Muller MJ, Amini L (2021a) How much automation does a data scientist want? ArXiv:2101.03970

Wang J, Li L, Zeller A (2019b) Better code, better sharing: on the need of analyzing jupyter notebooks

Wang J, Li L, Zeller A (2021b) Restoring execution environments of jupyter notebooks. In: 2021 IEEE/ACM 43rd international conference on software engineering, ICSE, IEEE, pp 1622–1633

Watson A, Bateman S, Ray S (2019) Pysnippet: Accelerating exploratory data analysis in jupyter notebook through facilitated access to example code. In: EDBT/ICDT Workshops

Zevin S, Holzem C (2017) Machine learning based source code classification using syntax oriented features. arXiv:170307638

Zhang AX, Muller M, Wang D (2020a) How do data science workers collaborate? roles, workflows, and tools. 2001.06684

Zhang G, Merrill MA, Liu Y, Heer J, Althoff T (2020b) Coral: code representation learning with weakly-supervised transformers for analyzing data analysis. arXiv:200812828

Zheng NN, Liu ZY, Ren PJ, Ma YQ, Chen ST, Yu Sy, Xue JR, Chen BD, Wang FY (2017) Hybrid-augmented intelligence: collaboration and cognition. Front Inf Technol Electr Eng 18(2):153–179

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Dhivyabharathi Ramasamy is a PhD student at the Department of Informatics, University of Zurich, Switzerland. She received her Master’s degree in Computer Science from the University of Zurich, Switzerland. Her research focuses on investigating how to support data scientists in exploring, implementing, and visualising the data science workflows using AI-based approaches.
Cristina Sarasua is a postdoctoral researcher at the department of informatics at the University of Zurich (Switzerland). Her research lies at the intersection of human-centered computing, data science and Semantic Web technologies. Prior to joining the University of Zurich, Cristina worked at the University of Koblenz-Landau (Germany) as a research and teaching assistant, where she received her PhD in Computer Science.

Alberto Bacchelli is an associate professor of Empirical Software Engineering of the Department of Informatics in the Faculty of Business, Economics and Informatics at the University of Zurich, Switzerland. He received his bachelor’s and master’s degrees in computer science from the University of Bologna, Italy, and his Ph.D. in Software Engineering from the Università della Svizzera Italiana, Switzerland.

Abraham Bernstein is a Full Professor of Informatics as well as director of the Digital Society Initiative — a university-wide initiative with more than 180 faculty members investigating all aspects of the interplay between society and the digitalization — at the University of Zurich (UZH), Switzerland. His current research focuses on various aspects of the semantic web, knowledge graphs, data mining/machine learning, data science, and collective intelligence. His work is based on both social science (organizational psychology/sociology/economics) and technical (computer science, artificial intelligence) foundations. Prior to joining the University of Zurich Mr. Bernstein was on the faculty at New York University and also worked in industry. Mr. Bernstein is a Ph.D. from MIT and holds a Diploma in Computer Science (comparable to a M.S.) from ETH Zurich.
Affiliations

Dhivyabharathi Ramasamy\textsuperscript{1} · Cristina Sarasua\textsuperscript{1} · Alberto Bacchelli\textsuperscript{1} · Abraham Bernstein\textsuperscript{1}

Cristina Sarasua  
sarasua@ifi.uzh.ch

Alberto Bacchelli  
bacchelli@ifi.uzh.ch

Abraham Bernstein  
bernstein@ifi.uzh.ch

\textsuperscript{1} Department of Informatics, University of Zurich, Zurich, Switzerland