Predicting File Completion Using Time Series Models: Embracing the Life-Cycle Nature of Software Development

Nastasja Stephanie Parschew[1][0009-0002-6492-3587] and
Saimir Bala[1,2][0000-0001-7179-1901]

[1] Humboldt-Universität zu Berlin, Berlin, Germany
[2] SAP Signavio, Berlin, Germany
nastasja.parschew@student.hu-berlin.de, saimir.bala@hu-berlin.de

Abstract. In software development, predicting the completion time of tasks is crucial for effective project management. However, due to the complexity of identifying process activities and thereby creating event logs for further analysis, traditional process prediction techniques cannot be readily applied to software data. A key challenge is that development activities are recorded as fine-grained file changes, spanning over multiple artifacts and connected to various other entities, such as the users who made the changes, the module membership, or other related links. In this paper, we present an approach to extract and analyze information from artifacts present in software repositories, allowing us to identify completion patterns exhibited in projects. Subsequently, we leverage time-series analysis to understand the evolution of these artifacts and predict their completion times based on file-level activity. We evaluate this approach against real-world data, showing its effectiveness and usefulness in predicting file completion times. Our work provides project managers with actionable insights into critical development areas, highlighting regions of unpredictability or potential delays, and improving decision-making.

Keywords: Time series analysis · Software process prediction · Software process mining · File completion

1 Introduction

Software development processes are a category of processes that are complex and challenging to control. A key difficulty faced by managers in effectively managing a software development project is the ability to anticipate potential issues that can influence the time required to complete certain tasks. Therefore, data-driven approaches that focus on mining the traces of the development process are of vital importance. These approaches can assist managers in getting a more comprehensive understanding of how the project is evolving. To achieve the best insights, managers are interested in both the as-is and to-be processes concerning the development of the software for which they are responsible.

Among the numerous data-driven techniques that help managers make decisions when dealing with highly complex and ambiguous data, process prediction methods stand out for their ability to provide a forecast about the completion times of activities. Unfortunately, these methods cannot readily be applied to *life-cycle* processes such as software development. When observing event logs from development, such as the ones extracted from version control systems (VCS), there are no explicit process activities that can be used to construct a traditional event log. Rather, activities must be inferred from low-level events that emerge from file changes of the artifacts recorded in the software repositories, such as GitHub, that are used by the VCS to track the software changes.

This paper proposes an approach to tackle the challenge of predicting task completion by leveraging time series analysis to help the manager understand the status of the process. We focus on the *case perspective* of the development, and our goal is to provide information on the quality and evolution of certain development cases, such as the development of a feature, the resolution of a bug, and so on. Our approach begins with identifying and extracting meaningful events from software repositories. We then introduce a method to map these events to specific process activities. By analyzing the time series data associated with these artifacts, we can predict task completion, offering valuable foresight into the software development life cycle. By doing so, our approach empowers project managers to detect areas of the software project that require attention, enhancing their ability to manage projects proactively.

The remainder of this paper is structured as follows. Section 2 describes the problem and discusses related work in the literature. Section 3 details our approach to the completion times of software development activities. Section 4 tests our approach against synthetic and real-world datasets. Section 5 discusses potential benefits and limitations. Section 6 highlights the key findings.

2 Background

2.1 Problem description

Mining the software process. The problem described in this paper falls under the umbrella of mining software repositories to identify and predict process activities. These activities are normally known to the project managers and other stakeholders, but their traces are normally scattered across various repositories and tool logs. As well, these repositories and logs are rarely process-aware. This means that there is no notion of cases or activities associated with events in their logs. Therefore, standard process mining algorithms are not readily applicable [19,4].

One problem of traditional process mining algorithms is the assumption that cases are mono-dimensional (i.e., their activities follow a sequence). This assumption does not fit the case of software development, where changes happen in different dimensions at once. For example, to track the progress of a software process with standard process mining, one can assume, as *case*, the sequence of

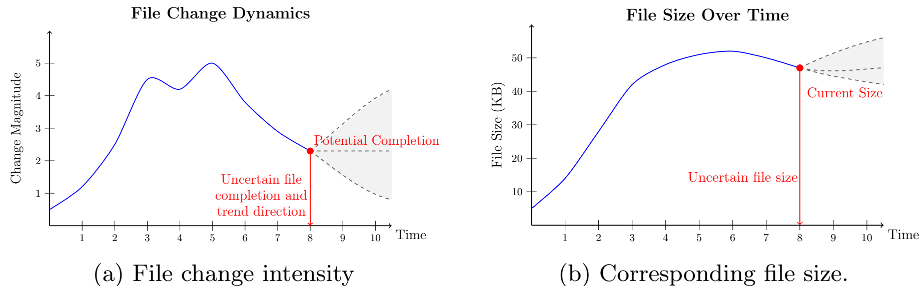
commits or the sequence of file changes, but not both. Object-centric process mining [1] overcomes this limitation by capturing related changes to each object. This allows for multi-dimensional process analysis techniques, which are more fit for analyzing software processes [11,16].

Yet, all these techniques still work with the assumption of an underlying process. That is, they all assume a so-called teleological process [23] to be in place. However, due to their ever-evolving nature [14], the software development process is rather a so-called life-cycle process [23]. As such, it is argued [12] that a suitable technique to analyze these processes is time-series analysis.

Analyzing the progress. Let us now look at the specific problem tackled in this paper through the following exemplary scenario.

A software development manager oversees a project. Their goal is not only to monitor how work is progressing, but also to support the team effectively by reallocating resources when needed, identifying bottlenecks early, and ensuring timely delivery. To this end, the manager relies on various tools such as Jira for planning, shared calendars for scheduling, and regular stand-up meetings where team members discuss their progress. However, each of these sources has significant limitations. Jira and related planning tools describe what *should* happen, not what *is* happening. On the other hand, progress communicated in meetings is often subjective, influenced by individual perceptions or incomplete information. To make informed, proactive decisions, the manager needs a more objective, data-driven view of the actual development process.

Fortunately, the team uses a VCS (Git in combination with GitHub), which continuously records detailed event logs of code evolution. These logs offer a rich source of behavioral data about the development process. However, applying traditional process mining techniques to these logs proves ineffective, as discussed earlier in this section. The core problem is that software development is not a well-structured process with clearly repeatable activities and cases, but rather a fluid, life-cycle process that unfolds differently for each task.



(a) File change intensity (b) Corresponding file size.

Fig. 1: (a): File change intensity during software development with uncertainty in future evolution. (b): Corresponding file size, which may grow, stabilize, or decrease depending on future changes.

To address this challenge, there is a need for techniques suited to life cycle processes. Therefore, we focus on analyzing development progress using time series extracted from version control data. Specifically, we consider the evolution of individual files that serve as meaningful indicators of a task's progression. As shown in Figure 1, one plot (a) captures the intensity and direction of *file changes over time*, while the other plot (b) illustrates the corresponding *growth of file size*. These views help interpret whether a task is actively being worked on, undergoing refinement, or nearing completion. For instance, a sudden decrease in change magnitude accompanied by file size stabilization could indicate that a file, and potentially the associated task, is being finalized. Through such patterns, we aim to support managers with data-informed insights into development activity. This brings us to our central research question:

RQ: How can we analyze progress in software development using time series extracted from version control systems?.

2.2 Related work

Previous work has extensively analyzed file-level metrics to assess software quality, change patterns, and defect susceptibility. Alshehri et al. [2] applied churn metrics in machine learning models to predict change-prone files, emphasizing their value for resource allocation. In related work, they also focused on identifying newly created files likely to change [3]. Bala et al. [5] introduced hidden co-evolution analysis to reveal non-hierarchical file dependencies, showing their impact on modularity and maintainability. Sas et al. [21] further linked frequent file co-changes to architectural flaws requiring refactoring. Additionally, Wang et al. [24] enhanced defect prediction by leveraging deep learning on semantic features extracted from file-level data.

While these studies contribute valuable insights, few integrate file evolution with process metrics or time-series analysis to support managerial decision-making. Bala et al. [5] used time-series methods to study development patterns but did not focus on predicting task progress. Choraś et al. [9,10] combined ARIMA models with process metrics to forecast Agile development indicators and built dashboards for managerial use, but did not focus on file-level granularity. Choetkiertikul et al. [7,8] offered predictive models for Agile sprints and story point estimation based on issue velocity and dependencies, remaining at a higher abstraction level. Similarly, Park and Song [17] and Rique et al. [20] addressed resource optimization through predictive analytics, yet did not consider the evolving dynamics of individual files. Most of these approaches rely on aggregated metrics, failing to capture granular trends at the file level.

Our work addresses this gap by combining file-specific time-series modeling with process metrics to predict the completion of individual files. This enables more fine-grained and actionable insights into development progress. As summarized in Table 1, our approach uniquely integrates file-level focus, time-series prediction, and practical managerial value, supporting improved task tracking.

Table 1: Comparison of Our Approach with Existing Work

Study	File-Level Focus	Time-Series Modeling	Completion Prediction	Managerial Use
Alshehri et al. [2,3]	✓	✗	✗	✗
Bala et al. [5]	✓	✓	✗	✗
Wang et al. [24]	✓	✗	✗(defects)	✗
Choetkiertikul et al. [7,8]	✗	✓	✓(story points)	✓
Choraś et al. [9,10]	✗	✓	✓(metrics)	✓
Park & Song [17], Rique et al. [20]	✗	✓	✗	✓
Our approach	✓	✓	✓	✓

3 Method

In this section, we outline our systematic approach to collecting, preprocessing, and analyzing data from multiple software repositories hosted on GitHub.

3.1 Overview of the Approach

Our approach is a structured, data-driven pipeline that transforms raw commit histories from GitHub repositories into predictive insights about file completion. The final output is a continuous target variable, *days until completion*, per file, enabling fine-grained tracking of development progress and informing managerial decision-making. The process unfolds in three major steps:

1. **Data Collection and Target Labeling**: We collect commit data at both the repository and file levels using the GitHub API. The data includes metadata, file changes, and commit histories. To ensure the robustness of our analysis, we select repositories based on the following criteria: *C1)* At least 2,000 commits, *C2)* Open-source and publicly accessible, *C3)* A minimum project age of one year. Based on these criteria, we curated a set of ten repositories for analysis.
2. **Feature Engineering and Model Input Construction**: From the collected commit data, we engineer a diverse set of features that describe temporal dynamics, file structure, and contributor behavior. These include rolling statistics, time intervals between commits, file size evolution, and categorical attributes such as file type and contributor identity. To build the predictive target, we define completion dates using three heuristics: stable line changes, file deletion, and prolonged inactivity. For each file with a completion date, we compute the number of days until completion for each commit.
3. **Model Training and Prediction**: We train supervised regression models including Random Forest [6], Gradient Boosting [15], and LightGBM [13] on the labeled dataset. Files without a valid completion label are excluded

from training but can still be analyzed qualitatively. We evaluate model performance using Mean Absolute Error (MAE) on a hold-out evaluation set and benchmark results against a linear regression model and a simple median baseline. To ensure generalization and avoid data leakage, we apply file-level splits and `GroupTimeSeries` cross-validation during model tuning.

3.2 Step 1: Data Preparation and Target Labeling

To model file-level completion within a software project, we begin by collecting and structuring the full commit history of each repository. Our goal is to transform raw, unstructured Git data into a clean, tabular dataset that can be used as input for machine learning models.

We use the GitHub REST API to extract: i) a complete list of commits in the repository (including SHA identifiers and timestamps), ii) detailed information for each commit (capturing added and deleted lines for each modified file), and iii) the full commit history of each file. For each commit, we record all affected file paths along with line-level changes and reconstruct the commit timeline for each file. This includes the file size after each commit, committer identity, and metadata such as commit intervals. The resulting structured dataset forms the foundation for feature engineering and model training.

To enable predictive modeling, we define the target variable as the number of days between a given commit and the moment a file is considered `complete`. Since Git data does not contain explicit labels for completion, we approximate completion using three strategies. First, if a file exhibits *minimal change* in at least three consecutive commits spanning over 14 days and then remains inactive for 30 or more days, we assign the last stable commit as the completion date. Second, if a file is *deleted* (indicated by its size dropping to zero), we label that commit as `complete` in the specific completion date and annotate the reason as `deleted`. Third, if a file has *not been modified* for a period that exceeds the 95th percentile of commit intervals in the repository (with thresholds clipped between 30 and 365 days), we assign its last commit as the completion point.

For files that meet any of these criteria, we compute the *days until completion* for each commit as the number of days until the identified completion date. Files that do not satisfy any of the conditions are considered incomplete and are excluded from supervised learning. The final dataset consists of one row per commit per file, including all derived features and possibly the target value. This dataset is now ready for the feature engineering and modeling steps that follow.

3.3 Step 2: Feature Engineering and Model Input Construction

With the structured commit data prepared, we construct a comprehensive set of features that capture both the temporal dynamics and structural characteristics of each file's evolution. The objective is to translate raw commit behavior into model-readable signals that can inform predictions about file completion.

For each commit in a file's history, we derive features based on commit activity, file size, and contributor behavior. These include rolling statistics over the most

recent commits, such as moving averages of added and deleted lines, recent file size growth, and the frequency of changes. To capture trends without introducing excessive noise, we compute most rolling features over the latest seven commits, which balances short-term relevance with temporal stability. In addition, we incorporate temporal features such as the number of days since the previous commit, the density of activity over the last 30 and 90 days, and the variation in time intervals between commits. These features help to characterize development intensity and highlight periods of stability or inactivity.

We also extract structural attributes of each file, including its extension, depth in the directory hierarchy, and classification (e.g., source code, configuration, or script). These features are used to distinguish core implementation files from peripheral ones. To model developer contributions, we track the identity of the committer and group infrequent contributors, defined as those responsible for fewer than 1% of a project's commits, into a shared `other` category. This information is encoded as a categorical variable to reflect patterns in team activity. To ensure a consistent input space, we filter out files with fewer than five commits, as these provide insufficient data for modeling. We also limit the dataset to files residing within core implementation directories (e.g., `src/`), excluding test, documentation, or build files that are less relevant to predictive modeling of file-level completion.

At the end of this process, we obtain a machine-learning-ready dataset where each row corresponds to a file-level commit, each column encodes a structured feature, and the target variable represents the number of days until file completion.

3.4 Step 3: Model Training and Prediction

To predict file completion times, we train and evaluate machine learning models on our engineered dataset, focusing on files with assigned completion dates. This step allows us to validate our approach by comparing the predictive performance of multiple algorithms. To validate our approach, our pipeline includes the training of various models. This step enables us to evaluate the different models on their ability to predict a file's completion times.

We split the labeled dataset (files with a "days until completion" target) into a training and evaluation set using an 80:20 ratio. Furthermore, we ensure random sampling while preventing commits from the same file from appearing in both sets to avoid data leakage. Files without already assigned completion dates are excluded from supervised modeling but may be used for exploratory analysis. Numerical features (e.g., file size, rolling statistics) are scaled while categorical features (e.g., file type, committer group) are one-hot encoded if the machine learning model requires it.

We train three primary models: Random Forest [6], Gradient Boosting [15], and LightGBM [13]. We selected these models due to their strong performance on tabular datasets, ability to capture non-linear feature interactions, and widespread use in practical machine learning applications. Including all three enables a comparison between traditional ensemble methods and optimized gradient-based approaches.

We compare our results with two baseline models: a linear regression model and a median baseline. The median baseline is computed to store the median `days_until_completion` of the train set and return this value for all further entries of the evaluation set. We expect this to be of the lowest accuracy. We tune the hyperparameters through Grid Search [22] with a 5-fold cross-validation. Furthermore, we used a `GroupTimeSeriesSplit` [18] to ensure that no commit data from the same file would be found in the cross-validation splits. For Random Forest, we optimize parameters such as the number of trees (50–300) and the maximum depth (5–30). For Gradient Boosting, we tune parameters including the learning rate (0.01–0.1), the number of estimators (100–400), and maximum depth (3–7). Through this process, we ensure well-calibrated models.

During training, models learn to predict the continuous target "days until completion" from the engineered features. For evaluation, we withhold the target labels (days until completion) in the evaluation set. We then generate predictions for each commit. Afterwards, we compare the predictions with the actual values using MAE, which measures average prediction error in days.

4 Evaluation

To evaluate our approach, we tested our pipeline on real-world data, focusing on the accuracy of file completion time predictions. This section details the experimental setup, data collection, and results, emphasizing the generalizability and practical utility of our method.

4.1 Experimental setup

To ensure reproducibility, we implemented a prototype [3] in Python. We conducted experiments on a system with an Intel Xeon CPU, 1TB of RAM, and Python 3.11. The core libraries include `pandas` (v2.2.3) for the dataset construction, `scikit-learn` (v1.6.1) for the implementation of the machine learning models, and `mlxtend` (v0.23.4) for grouped time series cross-validation using `GroupTimeSeriesSplit`. We evaluated our approach on 10 open-source GitHub repositories each, meeting criteria **C1** (>2,000 commits), **C2** (open-source), and **C3** (at least one year old), with files having assigned completion dates.

Our evaluation approach involves training models on the training set to predict the number of days until completion. For the evaluation set, we withheld completion labels, generated predictions, and compared those to the actual values. For each file in the evaluation set, the model predicts the number of days until completion. Predictions are then compared to actual values using Mean Absolute Error (MAE) as the primary evaluation metric.

4.2 Data collection

We collected commit and file data from ten GitHub repositories that satisfy our selection criteria **C1** through **C3**. These repositories, summarized in Table 2,

[3] https://github.com/mornningstar/completiontimes

vary widely in terms of their commit history size, age, programming languages, and application domains. The primary programming languages include Python, TypeScript, HTML, and Java, reflecting a range of development environments such as natural language processing, web frameworks, UI components, and data serialization libraries. This diversity ensures that our evaluation of the predictive models is robust across different types of software projects and development practices.

Table 2: Characteristics of Evaluated GitHub Repositories

Repository	Commits	Age (Years)	Primary Language
flairNLP/fundus	2,890	2.6	Python
khoj-ai/khoj	4,797	3.8	Python
vuejs/core	6,671	7.0	TypeScript
mozilla/addons-server	61,092	11.3	Python
fastapi/fastapi	5,806	6.5	Python
pallets/flask	5,437	15.2	Python
keras-team/keras	11,554	10.2	Python
tabler/tabler	2,995	7.3	HTML
google/material-design-lite	2,872	10.3	HTML
google/gson	2,112	10.2	Java

The repositories range from relatively young projects with a few thousand commits to mature, long-standing codebases with tens of thousands of commits. The wide variation in repository age (2.6 to 15.2 years) and commit volume (2,112 to 61,092 commits) provides a comprehensive testing ground for our approach, demonstrating its applicability and effectiveness in varied real-world settings.

4.3 Prediction results on real-world data

We report our prediction results in Table 3. It can be observed that LightGBM (LGBM) outperforms GradientBoosting (GBoost) and Random Forest (RForest) in 9 out of 10 projects. Furthermore, GradientBoosting outperformed Random Forest in most cases, although results vary depending on the project. The best performance was observed with the projects `flairNLP/fundus` (36.34 days with LightGBM) and `google/material-design-lite` (38.55 days with LightGBM), both having fewer than 3,000 commits, thus suggesting that smaller, younger projects are easier to predict. The most challenging projects to predict were `google/gson`, having an MAE over 200 days on average, and `fastapi/fastapi`, having an MAE of 266.36 days on average. The repository `google/gson` was one of the oldest ($>$ 10 years) and the only Java project in the set. The bad performance was likely due to complex commit patterns or language-specific characteristics. Surprisingly, `fastapi/fastapi` is the only project having better results with the Random Forest model and still being close to the worst LightGBM performance.

Table 3: Mean Absolute Error (MAE) in Days for Different Prediction Models Across Repositories. LinReg and Median are the baselines. Lower MAE means better predictive performance. Bold values highlight the best performance.

Repository	RForest	GBoost	LGBM	LinReg	Median
flairNLP/fundus	42.34	40.98	**36.34**	66.05	129.45
khoj-ai/khoj	124.20	115.69	**83.33**	131.55	128.92
vuejs/core	242.46	221.49	**139.25**	313.26	597.30
mozilla/addons-server	371.67	371.34	**347.14**	466.73	661.02
fastapi/fastapi	**266.36**	381.82	315.29	960.07	683.97
pallets/flask	221.03	304.21	**124.65**	668.95	417.32
keras-team/keras	208.47	187.82	**137.56**	249.73	383.85
tabler/tabler	250.18	287.49	**120.47**	307.22	518.79
google/material-design-lite	54.23	53.26	**38.55**	169.99	197.24
google/gson	294.45	296.07	**260.91**	1382.90	1541.89

To contextualize these results, we compared our approach to two baseline models: Linear Regression (LinReg) and Median Baseline (Median). There, we expected Linear Regression to perform better than the Median Baseline. Furthermore, we anticipated both to perform worse than our three main models (Random Forest, Gradient Boosting, and LightGBM). But this is not always the case. Only seven of the ten tried repositories validate the superiority of Linear Regression against a baseline algorithm, which predicts the training set's median days until completion. For the other three projects, the Median Baseline outperforms Linear Regression, which is unexpected. Nonetheless, throughout all ten repositories, the models depicted in Table 3 outperform any baseline algorithm. These results showcase the excellence of tree-based models concerning the variety of features and the possibility to learn from them.

5 Discussion

Unlike prior work that focused on aggregated metrics or issue-level, our approach works at the file level. This granularity allows project managers to identify specific files that are near or at completion, as well as those that are still under active development. By predicting days until a file's completion, project managers gain a forward-looking view into a project's state. This information supports them in evaluating timelines more precisely than commit count or higher-level project metrics alone allow.

Additionally, our approach combines temporal dynamics with structural properties, allowing a multi-faceted view of file evolution. By solely learning a project's patterns, it is not tailored to a single codebase or language, as demonstrated by our evaluation across ten open-source projects. As a result, the method is scalable and adaptable, requiring only minimal adjustments to be deployed in new development contexts.

Despite these strengths, limitations exist. Variability in commit practices, as seen with `google/gson`'s high MAE, may impact prediction accuracy. In such cases, large errors can signal ongoing refactoring, inconsistent development patterns, or frequent reopenings of seemingly finished files. These insights are still valuable for project managers. Extending file-level predictions to task-level outcomes also requires modeling task dependencies, a direction for future work.

Overall, our approach equips project managers with granular, interpretable, and adaptable insights, enhancing their ability to analyze the software process, anticipate bottlenecks, and drive project success.

6 Conclusion

In this paper, we presented a data-driven approach to predict file completion times using detailed commit histories. We implemented a prototype and analyzed data from multiple open-source repositories. By extracting meaningful features and applying machine learning models, especially LightGBM, our method consistently outperforms baseline models across diverse projects. These predictions provide valuable insights for project managers to monitor progress better and coordinate development efforts.

Future work can improve the approach by incorporating additional data sources such as issue trackers and communication logs, and by enabling real-time model updates. Furthermore, additional information on the task gathered from domain experts may be incorporated to best select the representative files, to which our time-series analysis applies. Enhancing the predictive accuracy and timeliness will further support proactive project management and improve software development workflows.

Acknowledgments. Supported by the Einstein Foundation Berlin EPP-2019-524.

References

1. van der Aalst, W.M.P.: Object-centric process mining: Dealing with divergence and convergence in event data. In: SEFM. Lecture Notes in Computer Science, vol. 11724, pp. 3–25. Springer (2019)
2. Alshehri, Y.A.: Predicting change in newly created files in a software product line project. Softw. Pract. Exp. **52**(12), 2499–2512 (2022)
3. Alshehri, Y.A., et al.: Can we predict the change in code in a software product line project? Journal of Software Engineering and Applications **13**(06), 91 (2020)
4. Bala, S., Mendling, J.: Monitoring the software development process with process mining. In: BMSD. Lecture Notes in Business Information Processing, vol. 319, pp. 432–442. Springer (2018)
5. Bala, S., Revoredo, K., de A. R. Gonçalves, J.C., Baião, F., Mendling, J., Santoro, F.M.: Uncovering the hidden co-evolution in the work history of software projects. In: BPM. Lecture Notes in Computer Science, vol. 10445, pp. 164–180. Springer (2017)
6. Breiman, L.: Random forests. Machine learning **45**, 5–32 (2001)

7. Choetkiertikul, M., Dam, H.K., Tran, T., Ghose, A., Grundy, J.: Predicting delivery capability in iterative software development. IEEE Trans. Software Eng. **44**(6), 551–573 (2018)
8. Choetkiertikul, M., Dam, H.K., Tran, T., Pham, T., Ghose, A., Menzies, T.: A deep learning model for estimating story points. IEEE Trans. Software Eng. **45**(7), 637–656 (2019)
9. Choras, M., Kozik, R., Pawlicki, M., Holubowicz, W., Franch, X.: Software development metrics prediction using time series methods. In: CISIM. Lecture Notes in Computer Science, vol. 11703, pp. 311–323. Springer (2019)
10. Choras, M., Springer, T., Kozik, R., López, L., Martínez-Fernández, S., Ram, P., Rodríguez, P., Franch, X.: Measuring and improving agile processes in a small-size software development company. IEEE Access **8**, 78452–78466 (2020)
11. Fahland, D.: Multi-dimensional process analysis. In: BPM. Lecture Notes in Computer Science, vol. 13420, pp. 27–33. Springer (2022)
12. Fahrenkrog-Petersen, S.A., Bala, S., Pufahl, L., Mendling, J.: Unraveling the never-ending story of lifecycles and vitalizing processes. In: EDOC Workshops. Lecture Notes in Business Information Processing, vol. 537, pp. 68–81. Springer (2024)
13. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: Lightgbm: A highly efficient gradient boosting decision tree. Advances in neural information processing systems **30** (2017)
14. Lehman, M.M., Ramil, J.F.: Software evolution and software evolution processes. Ann. Softw. Eng. **14**(1-4), 275–309 (2002)
15. Natekin, A., Knoll, A.: Gradient boosting machines, a tutorial. Frontiers in neurorobotics **7**, 21 (2013)
16. Nguyen, T., Bala, S., Mendling, J.: Multi-dimensional process analysis of software development projects. In: MODELSWARD. pp. 179–186. SCITEPRESS (2024)
17. Park, G., Song, M.: Optimizing resource allocation based on predictive process monitoring. IEEE Access **11**, 38309–38323 (2023)
18. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
19. Poncin, W., Serebrenik, A., van den Brand, M.: Process mining software repositories. In: CSMR. pp. 5–14. IEEE Computer Society (2011)
20. Rique, T., Perkusich, M.B., Dantas, E., Albuquerque, D., Gorgônio, K.C., de Almeida, H.O., Perkusich, A.: On adopting software analytics for managerial decision-making: A practitioner's perspective. IEEE Access **11**, 73145–73163 (2023)
21. Sas, D., Avgeriou, P., Kruizinga, R., Scheedler, R.: Exploring the relation between co-changes and architectural smells. SN Comput. Sci. **2**(1), 13 (2021)
22. Sun, Y., Ding, S., Zhang, Z., Jia, W.: An improved grid search algorithm to optimize SVR for prediction. Soft Comput. **25**(7), 5633–5644 (2021)
23. Van de Ven, A.H., Poole, M.S.: Explaining development and change in organizations. Academy of management review **20**(3), 510–540 (1995)
24. Wang, S., Liu, T., Nam, J., Tan, L.: Deep semantic feature learning for software defect prediction. IEEE Trans. Software Eng. **46**(12), 1267–1293 (2020)