An Automated Evaluation Approach for Jupyter Notebook Code Cell Recommender Systems

Selin Aydin[1,*], Dennis Mertens[1] and Ouyu Xu[2]

[1]Research Group Software Construction RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany
[2]ETH Zurich, Zurich 8092, Switzerland

Abstract

The reuse of code within Jupyter Notebooks is an often overlooked concept, resulting in the loss of valuable knowledge that could be retrieved from already created Notebooks. Presently, the reuse of Jupyter Notebooks is particularly impeded by the absence of suitable reuse approaches and tools.

This paper presents a code cell recommender system in the form of the JupyterLab extension, designated as JupyRecSys. Moreover, we propose an automatic evaluation framework in the form of the CL-tool CelRecEval, which allows developers to evaluate and compare the performance of cell recommender systems. The evaluation capabilities of CelRecEval are demonstrated by applying it to JupyRecSys. The resulting high metric scores demonstrate that JupyRecSys can correctly recommend and rank relevant cells. Moreover, the generation of more detailed performance reports enabled the identification of specific characteristics of code cells that negatively impact the performance of the cell recommendation system.

Keywords

Jupyter Notebook, Recommender System, Machine Learning

1. Introduction

The Jupyter Notebook (Notebook for short) has become a widely utilized tool in academia and industry for prototyping Python ML solutions due to their flexibility and interaction. Studies have demonstrated that code from other Notebooks is frequently reused through copy-paste. In particular, code for importing packages and visualizing data is often duplicated [1]. An analysis by Källen et al. of 2.7 million Notebooks on GitHub found that 70% of the code snippets contained are identical to others, differing only in whitespace, and nearly 50% of all Notebooks contain no unique code at all [2]. Currently, Notebooks lack reusability concepts. This issue has been discussed in various publications, and initial solutions have been proposed as best practices or JupyterLab extensions [3][4].

An initial challenge is to identify and locate relevant code from previous Notebooks. While search systems can be used for this purpose, they require manual intervention by the developer. The success of the search depends on the keywords used. An alternative approach would be to use a recommender system that suggests relevant cells from previous Notebooks to match the current cell. This would significantly reduce the effort required, as no action by the developer would be necessary.

A Notebook cell recommendation system (cell recommender for short) implements a special form of code recommendation, suggesting similar code cells instead of code completions. Cell recommenders have not yet been explored, and an evaluation method for them has not yet been developed.

The paper is structured as follows: Section 2 presents an overview of existing tools and methods to reuse Notebooks. In Section 3, we state the research goals and contributions of the paper. Then, in Section 4, we present a cell recommendation strategy. Next, Section 5 presents a general evaluation framework for cell code recommenders. Section 6 describes the implementation of the recommendation strategy and the evaluation framework. The setup and results of the evaluation are discussed in Section 7; the research questions are answered in Section 8. After presenting the threats to validity in Section 9, Section 10 concludes the paper and gives an outlook for potential future work.

2. Related Work

This section reviews recent tools and methods to improve the code reusability of Notebooks and their evaluation approaches.

JupySim, developed by Horiuchi et al., models Notebooks as directed acyclic graphs to identify the relational structures between code, data, and outputs [5]. The system has proven effective in identifying the most similar Notebooks based on user queries. However, its complex, detailed graph-based query construction may impede user adoption for regular Notebook reuse. Additionally, JupySim is currently only available as a separate web interface, suggesting the potential for a more integrated JupyterLab solution. Unfortunately, the authors do not provide an evaluation approach or results.

The Elyra code snippet JupyterLab extension represents a further significant addition to the Jupyter ecosystem [6]. Users can label specific code cells within any Notebook, save them to the global extension code database, and retrieve them via text or label queries. Despite Elyra's ease of integration and improved searchability, it relies heavily on manual user intervention. The necessity for users to repeatedly identify and label reusable code snippets could potentially be a source of frustration, particularly for those with extensive code bases.

While JupySim offers a sophisticated graph-based approach to identifying similar Notebooks, its complexity and standalone nature may not be optimal for everyday use. In contrast, Elyra, with its integrated JupyterLab interface, streamlines code snippet reuse but necessitates manual labeling and does not support the reuse of entire Notebooks.

QuASoQ 2024: 12th International Workshop on Quantitative Approaches to Software Quality, December 03, 2024, Chongqing China
*Corresponding author.
aydin@swc.rwth-aachen.de (S. Aydin); dennis.mertens1@rwth-aachen.de (D. Mertens); ouyuxu@ethz.ch (O. Xu)
https://github.com/d-mertens (D. Mertens)
0009-0006-1764-8091 (S. Aydin); 0009-0003-8076-0412 (D. Mertens); 0009-0006-4873-4187 (O. Xu)

A first step towards an automated solution is Typhon [7], an approach for recommending Notebook code cells based on Markdown text similarity. The authors evaluate Typhon twofold. First, they manually request and review recommendations for Markdown-code-pairs containing Matplotlib plot and chart code. For this, they add suitable Markdown-text to chosen code cells. The Markdown-text from these Markdown-code-pairs is then used to query recommendations. The authors evaluate a recommendation as *correct* if the recommended code is the same as in the query Markdown-code pair. The authors rated the resulting accuracy as *moderate*. Due to the evaluation setup, the generalizability and comparability are limited. It has to be noted that 30,93% of public Notebooks on GitHub do not contain a single Markdown cell [3]. Further, Typhon is not yet publicly available.

3. Research Goals and Contributions

Given the strengths and limitations of the existing approaches, our current work is focused on addressing these gaps by answering the following research questions (RQ):

RQ1: What can a cell recommendation strategy look like that makes suggestions to the developer during the programming of a cell?

RQ2: How can an approach for the automatic quantitative evaluation of the performance of cell recommenders look like?

By answering these questions, this paper makes the following contributions to improve the reusability of Notebooks:

- A cell recommendation strategy specialized for Python Notebooks implementing ML tasks.
- A general framework to quantitatively evaluate cell recommenders implementing this strategy.
- The JupyterLab extension JupyRecSys which implements the cell recommendation strategy.
- The CL-tool CelRecEval which implements this evaluation approach.
- The results of applying CelRecEval to evaluate the performance of the cell recommender JupyRecSys.

4. A Cell Recommendation Strategy

In this section, we present a recommendation strategy for cell recommenders. A cell recommender that suggests code based on similarity to a query cell is classified as a *content-based recommender*. Since it only provides the top-k most similar cells, we refer to it as a *top-k cell recommender*.

A common approach in such recommenders is to embed recommendation items in a *semantic vector space*. This has several advantages. First, it significantly reduces the complexity of the items, e.g., code, syntax, semantics, variables, symbols, etc. Further, a vector representation within a semantic vector space allows the recommender to efficiently determine similarities using various distance metrics, such as cosine similarity or Euclidean distance. In the vector space, the proximity of vectors directly reflects the degree of similarity between them. At the same time, the results remain interpretable because distances between vectors are easy to comprehend.

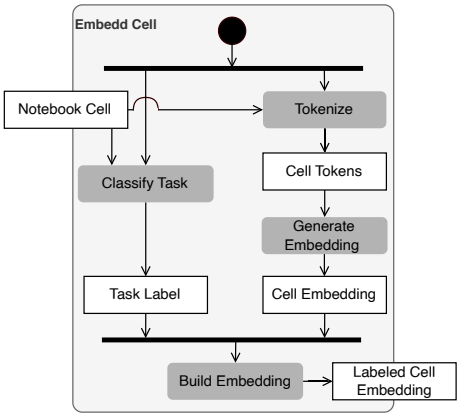

Figure 1: Process of embedding a Notebook cell

First, we explain how code cells are transformed into a vector representation in a semantic vector space, resulting in a *cell embedding* as output. Then, we present the process of *transforming and loading*, which stores cell embeddings in an appropriate database. Finally, we explain how recommendations are provided using the stored cell embeddings.

4.1. Embedding Cells

When code cells are transformed into vector representations and embedded in a vector space, they are particularly close to each other if they represent similar code. To further reduce the search space around a cell embedding and get more accurate results, each cell embedding is labeled with the ML task it implements.

The embedding process is shown in Figure 1 as a UML activity diagram. The input is a single Notebook cell. The process consists of two parts that are executed in parallel. The specific ML task implemented in a cell is *classified* in the left part, returning a *task label*, e.g. "data preprocessing" or "model training". The right part of the process consists of two actions. First, the code of the cell is *tokenized* into its elements (e.g., keywords, operators, identifiers), creating the *cell tokens*. Second, in the *generate embedding* action, the syntax and the semantic relationships between the tokens are analyzed and mapped into a vector space, returning a *cell embedding*. Finally, the task label and the cell embedding are composed to a *labeled cell embedding*.

4.2. Transforming and Loading Cells

To recommend code from previous Notebooks, the vector representations and required metadata of code cells must be stored in a dedicated database called *CelRec-DB*. A CelRec-DB must fulfill the following requirements:

- *Storage space*: the value and quality of the recommendations increase with the amount of available data for recommendation. Thus, the CelRec-DB must store the data in a scalable and efficient manner.
- *Data retrieval*: for recommendations, it is crucial that the stored data can be accessed quickly. Otherwise, the value is reduced if recommendations take too long.
- *Data representation*: the CelRec-DB has to support vectors and vector operations, ideally distance computations.

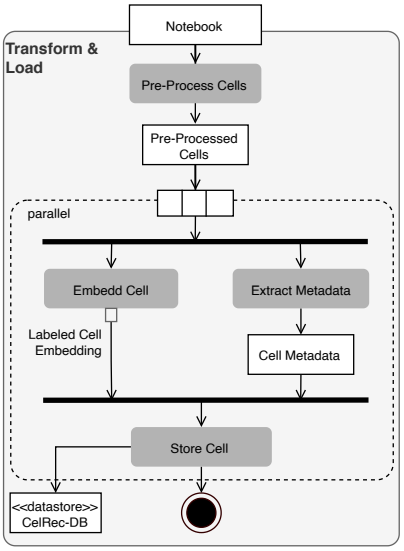

Figure 2: Process of transforming and loading cells in a database

Figure 2 depicts the process of storing the code cells of a Notebook in a CelRec-DB.

First, the Notebook is *pre-processed*. All code cells are extracted and cleaned, e.g., non-essential information like comments are removed. The resulting *pre-processed cells* are put into an ordered list.

Then, each pre-processed code cell undergoes two subprocesses. Each cell is transformed into a labeled cell embedding by applying the *embedd cell* process. In parallel, the *extract metadata* action returns metadata about the cell, such as its Notebook's name and the code it contains. Last, the labeled cell embedding and its associated metadata are *stored* in the CelRec-DB.

4.3. Recommending K Most Similar Cells

Given a Notebook containing one query cell, the recommendation process is depicted in Figure 3. First, the given Notebook goes through the *transform and load* process. This way, the query data is represented the same way, i.e., the query data is labeled and embedded in the same vector space as the recommendation data.

To recommend the k most similar code cells, the recommendation strategy takes advantage of the labels and the cell embeddings. First, all labeled cell embeddings in the database having the same label as the labeled query cell embedding are filtered to reduce the search space. Second, an *approximate nearest neighbor (k-ANN) search* using cosine similarity is performed to obtain the k most similar labeled cell embeddings to the labeled query cell embedding. Cosine similarity was chosen among other distance metrics because it provides a better-standardized comparability of vectors. Since the code snippet associated with a labeled cell embedding is stored as metadata, the associated code snippets of the k most similar cells are returned as recommendations.

5. A Framework for the Quantitative Evaluation of Cell Recommenders

After presenting the strategy for generating recommendations, we present in this section how a framework to quantitatively evaluate the performance of cell recommenders can be designed. By using this framework, we want to answer the following evaluation questions (EQ):

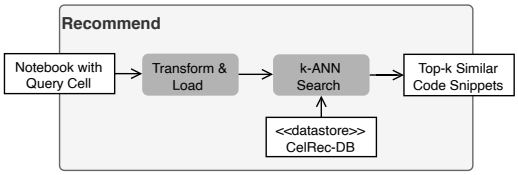

Figure 3: Recommendation strategy for given query Notebook

EQ1: What is the performance of the top-k cell recommender?

EQ2: How does its performance vary with increasing lines of code in a query cell?

The first question concerns performance in general. The second question is specific to a developer receiving recommendations while programming and wanting to write more lines of code in a cell. This would also show how much code is needed to obtain relevant recommendations.

5.1. Evaluation Methodology

Code cell recommendation for Notebooks is a special case of code recommendation. Code recommenders, in general, usually suggest code for code completion. There are two main strategies for evaluating code recommenders: *partial code reduction* and *user studies* [8].

In the former, code snippets are taken, and the last lines are removed, mimicking that the developer has started typing code and expects a recommendation. Then, given the code recommendation, it is checked whether it matches the removed code. This allows for the analysis of the recommender's performance. If this is done regularly, it can be quickly decided if a recommendation strategy or data change has had a positive or negative impact.

User studies, on the other hand, can verify that developers perceive the recommendations as *relevant*. However, they require much time and effort and are difficult to generalize and replicate.

Consequently, the partial code reduction evaluation is more efficient in checking whether a change in the recommendation strategy improves its performance. For this reason, we used a partial code reduction strategy with generated *evaluation data* consisting of *query* and *recommendation data*. We added noise to the recommendation data to "confuse" the recommendation strategy and thereby test its robustness.

5.2. Evaluation Data

Before recommendations can be requested, the database must be filled with recommendation data and query data for which recommendations will be given.

The conceptual idea of generating the query and recommendation datasets is illustrated in Figure 4. In the following, we will describe the generation process for each dataset in detail.

Query Dataset: Since we want developers to receive recommendations as they type, query cells with different numbers of lines of code per cell should be contained in the query dataset.

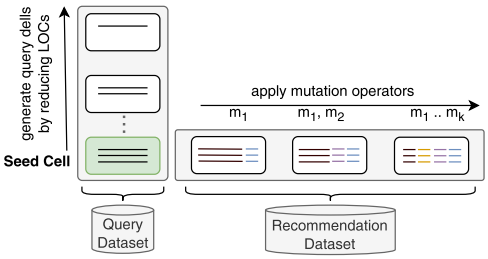

Figure 4: Generation of the query and recommendation datasets based on seed cells

To this end, the generation starts with a set of unique Notebook code cells $\mathbb{S}$, referred to as *seed cells*.

The number of lines in each seed cell is gradually reduced in the generation process, starting with the last line until only the first line of code is left. Empty lines are ignored. Hence, for a seed cell with n lines of code, n query cells are generated. For each seed cell s, the set QC_s, consisting of the seed cells and all generated query cells, is added to the query dataset $\mathbb{QD}$.

Recommendation Dataset: Since a top-k cell recommender suggests the top-k cells that are most similar, the recommendation dataset must have k recommendation cells with different degrees of similarity for each seed cell.

The process to generate these recommendation cells has to ensure that the different degrees of similarity conform to the expected order of the top-k recommendations. To generate similar recommendation cells from a given seed cell $s \in \mathbb{S}$, mutation operators are applied. Therefore, a set of k ordered mutation operators $\{m_1..m_k\}$ must be defined. Using these mutation operators, the generation process is as follows:

- **Step 1**: apply the mutation operator m_1 to the seed cell s.
- **Step i**, $2 \leq i \leq k$: apply the mutation operator m_i on the recommendation cell generated in step i-1.

Since each generation step applies one more mutation operator on the original seed cell, the similarity of the generated recommendation cell decreases step by step.

5.3. Assessment of Recommendations

Our assessment approach is based on the following hypothesis: a perfect top-k cell recommender recommends the mutated recommendation cells according to the number of mutation operators applied. Thus, the first recommendation would be the recommendation cell with one mutation operator applied, the second recommendation with two, and so on.

Therefore, a recommendation to a query cell is considered *relevant* if it results from applying mutation operators to the original seed cell.

Metrics are used to evaluate a top-k cell recommender's performance. Some metrics use a *relevance classification* of the given recommendations. Others use a *rating score* and a *rating threshold*.

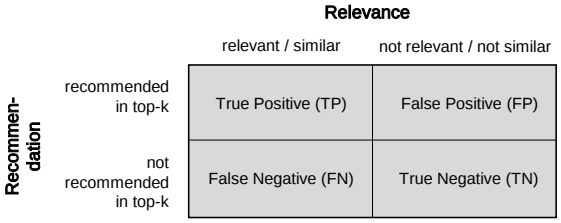

Figure 5: Relevance confusion matrix for cell recommendations

Relevance Classification

Figure 5 shows a confusion matrix representing the classification schema for recommendations. It defines the classes True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN).

For each top-k recommendation for a query cell, its relevance class must be decided according to our hypothesis, leading to the following classification rules:

1. If a recommendation is a mutated version of the seed cell, it is considered relevant and classified as TP.
2. If a mutated version of the seed cell is not listed in the top-k recommendations, then the missing recommendation is classified as FN.
3. If the top-k recommendation list contains a recommendation cell not generated by applying mutation operators to the seed cell, it is not considered relevant and is classified as FP.
4. If a recommendation is not generated from the seed cell and does not appear in the top-k recommendations, it is classified as TN.

Relevance Rating

To rate the relevance of a recommendation r, a rating function needs to be defined that maps r to the values of a scale. This rating scale must specify values for each relevant (mutated) recommendation and include at least one value for irrelevant recommendations. The values must be different for each recommendation resulting from a different number of applied mutation operators so that the ranking order of the recommendations can be considered.

A top-k cell recommender requires a rating scale with at least k+1 values where k values are above the rating threshold.

The following section describes the selected performance metrics based on this classification scheme and relevance rating.

5.4. Selected Performance Metrics

Commonly used metrics for top-k recommender are precision@k, recall@k, AP@k and nDCG@k [9]. In this section, we present the adoption of these metrics to evaluate a top-k cell recommender.

precision@k: measures how many recommendations of the top-k positions are relevant. For a query cell $q \in \mathbb{QD}$ and the set TP_q consisting of the relevant cells in the top-k recommendations for this query cell, is it calculated as follows:

$$precision@k_q = \frac{|TP_q|}{k}$$

recall@k: measures the share of relevant cells that are also contained in the top-k recommendations. Its value is calculated as follows:

$$recall@k_q = \frac{|TP_q|}{|TP_q \cup FN_q|}$$

where the set $TP_q \cup FN_q$ consists of recommendations considered relevant for the query cell $q \in \mathbb{QD}$.

F1@k: combines the metrics precision@k and recall@k in one metric and weights them equally. It is calculated this way:

$$F1@k_q = 2 \times \frac{precision@k_q \times recall@k_q}{precision@k_q + recall@k_q}$$

AP@k: measures the average of the precision values at different cut-off points in the top-k recommendations. It evaluates the accuracy of the ranking order of the top-k recommendations for a query cell $q \in \mathbb{QD}$.

It is calculated as follows:

$$AP@k_q = \frac{\sum_{i=1}^{k} precision@i_q \times relevance(r_i)}{TP_q}$$

where the function $relevance(r_i)$ for the recommendation r_i at rank i is defined as follows:

$$relevance(r_i) = \begin{cases} 1, & \texttt{if } rating(r_i)_q \geq z. \\ 0, & \texttt{otherwise}. \end{cases}$$

The rating function $rating(r_i)_q$ of a recommendation r_i has to be defined as described in Section 5.3. If the resulting rating score is above the defined rating threshold z, the recommendation is considered relevant.

$AP@k_q$ results in the highest score (1) if relevant recommendations (i.e., the mutations) are ranked higher than irrelevant recommendations in the top-k positions. However, the ranking order of the relevant recommendations does not matter. The score is the lowest (0) if no relevant recommendations are shown in the top-k positions.

nDCG@k: In addition to the AP@k, the Normalized Discounted Cumulative Gain also evaluates the ranking order of the recommendations based on a non-binary relevance rating. This also enables an evaluation of the ranking order of the relevant recommendations (i.e., recommendations r with $rating(r)_q \geq z$).

Given a list of recommendations R_q for a query cell $q \in \mathbb{QD}$, DCG is calculated based on the defined rating function $rating(r_i)_q$ for each recommendation $r_i \in R_q$ for a rank $i \in [1, ..., |R_q| = k]$ as follows:

$$DCG@k_q = \sum_{i=1}^{k} \frac{2^{rating(r_i)_q} - 1}{log_2(i+1)}$$

Each DCG@k value per query cell is normalized. This allows us to compare the accuracy between different recommenders, even returning recommendation lists of different lengths:

$$nDCG@k_q = \frac{DCG@k_q}{max_\pi DCG_\pi@k_q}$$

where $max_\pi DCG_\pi@k_q$ represents the DCG for an optimal recommendation list R_q^+ for the query cell $q \in \mathbb{QD}$. R_q^+ is created from R_q with a permutation π that reorders the recommendations $r \in R_q$ such that they are in a descending ranking order based on their assigned rating $rating(r)_q$ for the query cell q.

$nDCG@k_q$ results in the highest score (1) for a query cell q if the recommendations are in the ranking order of their relevance based on the assigned ratings.

Mean: Let $\mathbb{M}$ = {precision@k, recall@k, AP@k, nDCG@k } be the set of the selected performance metrics. This metric computes the overall mean for each used performance metric $m \in \mathbb{M}$ for a given set of query cells $Q \subseteq \mathbb{QD}$ as follows:

$$mean_m@k(Q) = \frac{1}{|Q|} \sum_{q \in Q} m@k_q$$

5.5. Procedure

To calculate and present these metrics, the following steps are performed, given a set $\mathbb{S}$ of seed cells:

1. Generate the query dataset $\mathbb{QD}$ and the recommendation dataset based on $\mathbb{S}$.
2. For each query cell $q \in \mathbb{QD}$:
   a) Request recommendations for q.
   b) Classify the recommendations according to the classification rules.
   c) Compute all metrics $m \in \mathbb{M}$ for q.
3. For each metric $m \in \mathbb{M}$:
   a) Compute $mean_m@k(\mathbb{QD})$ on all values of m.
4. For each seed cell $s \in \mathbb{S}$ and for each $m \in \mathbb{M}$:
   a) Compute $mean_m@k(QC_s)$ on all values of m.
5. Create the performance report.

6. Implementing the Strategy and the Evaluation Framework

To demonstrate the application of the evaluation framework to a top-k cell recommender, we developed JupyRecSys, implementing the cell recommendation strategy presented in Section 4 and the CelRecEval tool to automate the presented evaluation framework. The implementations and adjustments to the recommendation strategy and the evaluation framework are described below.

6.1. JupyRecSys - A Top-3 Cell Recommender

The top-3 cell recommender JupyRecSys is implemented as a Python library, which is integrated into a JupyterLab extension. It consists of a back- and frontend; its user interface is shown in Figure 6. The backend provides REST endpoints to upload Notebooks to the CelRec-DB or to request recommendations. To access the CelRec-DB, a dedicated database service is provided, which offers all most often needed queries to facilitate the search for similar cells.

We used existing tools and technologies to implement some steps in the recommendation strategy. To implement the *embedd cell* process, we applied our cell labeling tool JupyLabel [10] to get the cell's task label. Further, we used the pre-trained CodeBERT model [11] to analyze the cell tokens and map them into a vector space in the implementation of the *generate embedding* action. Finally, we used the specialized vector database Milvus DB [12] to set up the CelRec-DB and applied its optimized k-ANN search method that supports cosine similarity as a standard feature.

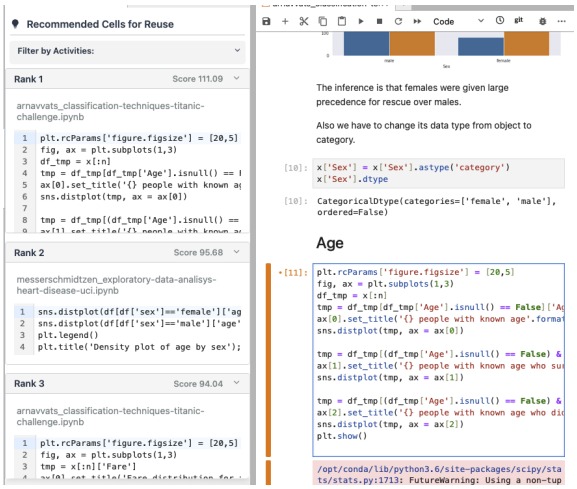

Figure 6: Screenshot of JupyRecSys UI with provided code cell recommendations

6.2. CelRecEval - A Cell Recommender Evaluation Tool

CelRecEval implements the cell recommender evaluation framework, especially for JupyRecSys. It is provided as a Python CL-tool, allowing the user to configure various parameters such as the number of recommended cells (top-k), the number of recommended cells per seed cell, or the metric selection. In addition, the report's output format, e.g., an Excel spreadsheet, can be configured.

As the generation of the evaluation data depends on the cell recommender's k-value and we wanted to evaluate the top-3 cell recommender JupyRecSys, three mutation operators for the generation of the recommendation data are implemented:

- M1: Only the variable names are changed. This is a common case when rewriting similar code.
- M2: The variable names are changed, and comment lines separate the lines of the query cell. This case is especially interesting for recommendation strategies, which also consider comments.
- M3: The variable names are changed, comment lines separate the lines of the query cell, and the code lines of the query cell are duplicated and inverted.

The resulting cell mutations, especially the ones generated by the M3 oerator, which introduces a lot of noise by generating nonsense code, allow us to make statements about the robustness of the recommendation strategy.

The following code illustrates the application of the mutation operators on an example seed cell:

```
lr = LogisticRegression()
lr.fit(X_train,y_train)
```

Listing 1: Example seed cell

After applying all three mutation operators, the following code cell is generated as one of the recommendation cells:

```
new_lr = new_LogisticRegression()
noissergeRcitsigoL_wen = rl_wen
# Additional comment line
new_lr.fit(new_X_train, new_y_train)
)niart_y_wen ,niart_X_wen(tif.rl_wen
```

Listing 2: Recommendation cell with M1-M3 applied

CelRecEval implements the cell relevance classification rules based on the presented confusion matrix (Section 5.3). As relevance rating, the following scores are assigned to the top-3 recommendations, based on how the recommendation is related to the original seed cell of the query cell: 5 (M1 applied), 4 (M1 & M2 applied), 3 (M1, M2 & M3 applied) and 1 (not related to query cell). Consequently, the used relevance threshold is $z = 3$. Further, CelRecEval implements the selected performance metrics accordingly.

The parameters must be changed to apply CelRecEval to other top-k cell recommenders. Further, the mutation operators and the REST endpoints for uploading query and recommendation data and requesting recommendations must be adapted accordingly. If a different CelRec-DB is used, an adapter for deleting the data has to be provided since each evaluation has to start with an empty database.

7. Evaluation

For the evaluation, we selected 114 unique seed cells from the Notebooks provided by KGTorrent [13]. Using CelRecEval, a query dataset with 810 cells and a recommendation dataset with 342 cells were generated. The 810 recommendation lists returned by JupyRecSys were evaluated, and the performance report was generated. Two sets of mean performance scores (MPS) were calculated:

- MPS1: All query cells' mean scores of all performance metrics $m \in \mathbb{M}$ were computed. $mean_m@k(\mathbb{QD})$
- MPS2: For each seed cell $s \in \mathbb{S}$, the mean performance scores of all performance metrics $m \in \mathbb{M}$ for all its generated query cells (QC_s) were computed. $\forall s \in \mathbb{S} : mean_m@k(QC_s)$

This way, we can gain insights into the overall performance, analyze performance across different code cell types, and identify difficulties with specific seed cells.

In the following, we interpret the obtained performance metric scores according to the evaluation questions EQ1 and EQ2 presented in Section 5.

7.1. EQ1: Performance of the Recommendation Strategy

Table 7.1 depicts the mean performance scores of all query cells (MPS1).

The overall mean precision@3, recall@3, and F1@3 scores are all equal, with a value of 0.8697. This is due to the specific characteristics of the evaluation setup. As each recommendation list contains exactly three recommendations, the number of potentially relevant recommendations is also 3. Therefore, precision@3 and recall@3 have the same denominator ($|TP_q \cup FN_q| = 3$) and also share the same numerator ($|TP|$). If precision@3 and recall@3 are equal, their harmonic mean is also equal, resulting in the same value for the F1@3 score.

Moreover, the mean AP@3 achieves an even higher, remarkable value of 0.9579. This highlights the strategy's effectiveness in prioritizing relevant recommendations at the top of the recommendation lists. In this evaluation, a query cell's mutated and relevant versions are consistently ranked higher than other irrelevant cells.

The higher AP@3 score indicates that while the recommendations are correctly ranked, some relevant cells are

MPS1: Mean Performance Scores				
precision@3	recall@3	F1@3	AP@3	nDCG@3
0.8697	0.8697	0.8697	0.9579	0.817

Table 1
Mean performance scores of all query cells ($\forall m \in \mathbb{M} : mean_m@k(\mathbb{QD})$)

not always included in the recommendation lists. By analyzing the MPS2 scores (see Table 7.2), we observe that the following cell characteristics negatively impact the recommendation strategy's performance :

- **For-loops:** The three seed cells with the lowest precision@3, recall@3 and F1@3 scores are the cells s_{76}, s_4, and s_{40}. All three cells contain a for-loop with data operations. In particular, if a query cell only contains the first line of a for-loop, this leads to matches with all cells that contain a for-loop, as these usually only differ in the variable names.
- **Multiple tasks:** Seed cell s_8 achieved only moderate results. A closer look revealed that this cell implements multiple tasks: dependency import, model prediction, and model evaluation. Considering only the first few code lines, the recommender suggests other dependency import code cells.
- **Many code lines:** It could be observed that the strategy's performance is lower for seed cells having many lines of code, e.g., $SLOC_{s4} = 12$.

7.2. EQ2: Impact of SLOCs to Performance Scores

If a seed cell contains precisely one code line, one query cell and three mutated recommendation cells are created. For this query cell, the recommendations given are perfect, and the performance scores are optimal. This was the case for 14 out of 114 seed cells.

Our analysis of the performance results indicates a significant correlation between the number of code lines and the strategy's performance. The generated query cells, having fewer lines than their common seed cell, sometimes exhibit higher similarity with recommendation cells with similar numbers of lines instead of the recommendation cells that would ultimately be relevant to the developer. This phenomenon can lead to erroneous recommendations, lowering the average metric result for the specific seed cell. Therefore, it can be concluded that the more lines a developer enters, the more accurate the recommendations will become.

Nevertheless, even the lowest-performing seed cell achieves an nDCG@3 value of at least 0.5, indicating that approximately half of the relevant recommendations are still recommended in the correct ranking order within the top-3 ranks.

8. Discussion

In response to research question RQ1, we proposed a cell recommendation strategy, which we implemented in the JupyterLab extension JupyRecSys. The transformation of code cells into a semantic vector space enables the efficient computation of similarities between a query cell and a large number of recommendation cells. Applying an automatic cell task classifier could further reduce the search space within the semantic vector space. The evaluation results in Section 7 highlight the effectiveness of the proposed cell recommendation strategy.

To answer RQ2, we designed and implemented an evaluation framework in the CL-tool CelRecEval.

We could have generated the evaluation data with identical query-recommendation pairs. However, this would have only shown that the similarity calculation works and would be a rather simplistic evaluation for a top-k cell recommender, where ranking also plays a role. This would not reflect real-world usage.

In practice, developers would receive recommendations as they type rather than when they have finished writing a code cell. By introducing cells with different numbers of lines of code and using mutation operators, we were able to identify weaknesses such as *for-loops*, *multiple tasks*, and *many lines of code*. Without automated evaluation, it would have been necessary to manually check all query cells to identify these issues.

Furthermore, CelRecEval can be re-executed with the same setup so that we can evaluate in the future whether changes to the implemented recommendation strategy eliminate the weaknesses mentioned above.

Consequently, CelRecEval enables developers to identify the strengths and weaknesses of cell recommenders and compare different ones.

9. Threats to Validity

Internal Validity

In this paper, we automatically evaluated the performance of a cell recommendation strategy using our definition of *relevancy* of a recommendation. When evaluating recommenders, users usually evaluate a recommendation's relevancy in a so-called online evaluation. However, the impact of this threat should be limited, as we consider the code's similarity and the performed task implemented in a cell as criteria for recommendation.

External Validity

The evaluation framework can be adapted with minor adjustments to other top-k cell recommenders. It allows the automatic generation of evaluation data suitable for the selected k. Currently, the evaluation performance results are used to compare different versions of JupyRecSys against each other. Different values for k and different sets of mutation operators may hinder comparability among recommenders.

10. Conclusion and Future Work

This paper has two main contributions. First, we present a strategy for Jupyter Notebook code cell recommendation.

MPS2: Mean Performance Scores for selected Seed Cells (sorted by nDCG@3)						
Seed Cell ID	SLOC	precision@3	recall@3	F1@3	AP@3	nDCG@3
48	1	1	1	1	1	1
70	1	1	1	1	1	1
103	1	1	1	1	1	1
...	...	...	...	...	...	...
8	6	0.7143	0.7143	0.7143	0.7143	0.6514
...	...	...	...	...	...	...
76	20	0.5333	0.5333	0.5333	0.7833	0.6443
4	12	0.5385	0.5385	0.5385	0.7436	0.6406
40	2	0.6667	0.6667	0.6667	0.75	0.5001

Table 2
Mean performance scores of all query cells QC_s generated from all seed cells ($\forall QC_s s \in \mathbb{S}, \forall m \in \mathbb{M} : mean_m @k(QC_s)$)

Second, we present a framework for automatically evaluating such cell recommenders. The framework allows users to customize the generation of evaluation data, select performance metrics, and access external tools, such as the cell recommender or its database. Both the recommendation strategy and the evaluation framework are implemented as tools: the strategy as the JupyterLab extension JupyRecSys cell recommender and the evaluation framework as the Python CL-tool CelRecEval.

To demonstrate the application of CelRecEval to a cell recommender, the framework was adapted to JupyRecSys. The evaluation results demonstrate that JupyRecSys exhibits high performance across all metrics, effectively delivering relevant code cells in a near-optimal order as the developer types code into the cell. Given its high metric scores, it is particularly well-suited to enhancing the reusability of Notebooks. In addition to evaluating general performance, we can also make statements about how the cell recommender performs while a developer is typing into a code cell. Further, CelRecEval allows us to identify some strengths and weaknesses of JupyRecSys.

As part of our future research, we aim to extend CelRecEval with more performance evaluation metrics to consider more recommender quality attributes, such as diversity or confidence.

Furthermore, the recommendation strategy of JupyRecSys could be improved by including Markdown text in the similarity analysis. An online evaluation would give a more accurate picture of the performance of the cell recommender, as users would evaluate the relevance of the recommendations in a real-world setting. A user study with developers could also provide essential insights into the usability of this recommender.

All software artifacts, including JupyRecSys, CelRecEval and the evaluation data are available on Zenodo [14].

References

[1] N. Ritta, T. Settewong, R. G. Kula, C. Ragkhitwetsagul, T. Sunetnanta, K. Matsumoto, Reusing My Own Code: Preliminary Results for Competitive Coding in Jupyter Notebooks, in: 2022 29th Asia-Pacific Software Engineering Conference (APSEC), 2022, pp. 457–461.

[2] M. Källén, T. Wrigstad, Jupyter Notebooks on GitHub: Characteristics and Code Clones, The Art, Science, and Engineering of Programming 5 (2021).

[3] J. F. Pimentel, L. Murta, V. Braganholo, J. Freire, Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks, Empirical Software Engineering 26 (2021) 65.

[4] A. Rule, I. Drosos, A. Tabard, J. D. Hollan, Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding, Proc. ACM Hum.-Comput. Interact. 2 (2018).

[5] M. Horiuchi, Y. Sasaki, C. Xiao, M. Onizuka, JupySim: Jupyter Notebook Similarity Search System., in: EDBT, 2022, pp. 2–554.

[6] Elyra Team, Code Snippets - Elyra 3.15.0 documentation, https://elyra.readthedocs.io/en/v3.15.0/user_guide/code-snippets.html, 2022. [Acc. 19-Sep-2024].

[7] C. Ragkhitwetsagul, V. Prasertpol, N. Ritta, P. Saewong, T. Noraset, M. Choetkiertikul, Typhon: Automatic Recommendation of Relevant Code Cells in Jupyter Notebooks, 2024.

[8] S. Proksch, S. Amann, S. Nadi, M. Mezini, Evaluating the evaluations of code recommender systems: A reality check, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 111–121.

[9] Y.-M. Tamm, R. Damdinov, A. Vasilev, Quality Metrics in Recommender Systems: Do We Calculate Metrics Consistently?, in: Proceedings of the 15th ACM Conference on Recommender Systems, RecSys '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 708–713.

[10] M. Perez, S. Aydin, H. Lichter, A Flexible Cell Classification for ML Projects in Jupyter Notebooks, 2024. URL: https://arxiv.org/abs/2403.07562. arXiv:2403.07562.

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained Model for Programming and Natural Languages, 2020. URL: https://arxiv.org/abs/2002.08155. arXiv:2002.08155.

[12] Milvus, The High-Performance Vector Database Built for Scale, https://milvus.io/, 2024. [Acc. 13-Sep-2024].

[13] L. Quaranta, F. Calefato, F. Lanubile, KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021.

[14] S. Aydin, D. Mertens, O. Xu, Zenodo: An Automated Evaluation Approach for Jupyter Notebook Code Cell Recommender Systems - Software Artifacts, https://doi.org/10.5281/zenodo.13836922, 2024.