A framework and toolkit for testing the correctness of recommendation algorithms

A Framework and Toolkit for Testing the Correctness of Recommendation Algorithms

LIEN MICHIELS and ROBIN VERACHTERT, Froomle, Belgium and University of Antwerp, Belgium
ANDRES FERRARO*, McGill University, Canada
KIM FALK, Shopify, Canada
BART GOETHALS, University of Antwerp, Belgium, Monash University, Australia, and Froomle, Belgium

Evaluating recommender systems adequately and thoroughly is an important task. Significant efforts are dedicated to proposing metrics, methods and protocols for doing so. However, there has been little discussion in the recommender systems' literature on the topic of testing. In this work, we adopt and adapt concepts from the software testing domain, e.g., code coverage, metamorphic testing, or property-based testing, to help researchers to detect and correct faults in recommendation algorithms. We propose a test suite that can be used to validate the correctness of a recommendation algorithm, and thus identify and correct issues that can affect the performance and behavior of these algorithms. Our test suite contains both black box and white box tests at every level of abstraction, i.e., system, integration and unit. To facilitate adoption, we release `RecPack Tests`, an open-source Python package containing template test implementations. We use it to test four popular Python packages for recommender systems: `RecPack`, `PyLensKit`, `Surprise` and `Cornac`. Despite the high test coverage of each of these packages, we find that we are still able to uncover undocumented functional requirements and even some bugs. This validates our thesis that testing the correctness of recommendation algorithms can complement traditional methods for evaluating recommendation algorithms.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Information systems → Recommender systems**; • **Human-centered computing → Collaborative filtering**.

ACM Reference Format:
Lien Michiels, Robin Verachtert, Andres Ferraro, Kim Falk, and Bart Goethals. 2022. A Framework and Toolkit for Testing the Correctness of Recommendation Algorithms. 1, 1 (June 2022), 47 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Recommender systems are widely used to help people discover products or content they might like, for example, on e-commerce websites or online streaming platforms. Their broad adoption makes it very important to adequately and thoroughly evaluate the performance of these systems. Evaluating a recommender system is a difficult task that can be accomplished in many ways [76]. The most often encountered experiment type in the literature is offline evaluation [6, 9, 76]. Here, a dataset is split into a training, validation and test dataset according to some procedure and the performance of the recommendation algorithm is compared to that of several 'baselines' according to some

*Currently at Pandora-SiriusXM

Authors' addresses: Lien Michiels, lien.michiels@uantwerpen.be; Robin Verachtert, robin.verachtert@froomle.com, Froomle, Belgium and University of Antwerp, Antwerp, Belgium; Andres Ferraro, andresferraro@acm.org, McGill University, Montréal, Canada; Kim Falk, Kim.falk.jorgensen@gmail.com, Shopify, Canada; Bart Goethals, bart.goethals@uantwerpen.be, University of Antwerp, Antwerp, Belgium and Monash University, Melbourne, Australia and Froomle, Antwerp, Belgium.

XXXX-XXXX/2022/6-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

metrics. Offline evaluation is frequently criticized, for example, for its poor correspondence to the performance of recommender systems in the real world [5, 33, 34]. Also, small changes in setup can lead to vastly different results and performance rankings between algorithms [3, 18, 19, 58]. Because evaluating recommender systems adequately and thoroughly is of great importance, the community dedicates significant efforts towards proposing new evaluation metrics [73] or data splits [35, 72] that do not suffer from the same shortcomings as their predecessors. However, the practice of evaluating a recommendation algorithm only by comparing its performance to that of other recommendation algorithms, using some evaluation metric that reports a single aggregate value across users on a held-out dataset, remains a constant throughout these works. We argue that an additional, complementary perspective on the evaluation of recommender systems is warranted. Implicitly, the practice of offline evaluation trusts that the recommendation algorithms evaluated are well-designed and implemented correctly, and are therefore robust, reliable and free of errors. However, we will show that this is not guaranteed. In software engineering, significant effort is spent designing test cases to validate that code is indeed reliable and free of errors [51]. Designing such test cases is considered a software engineering best practice, and is even included in the ACM/IEEE-CS Software Engineering Code of Ethics [24].

In recent years, software testing research has proposed new paradigms and tools for the automated testing of complex software systems, and machine learning systems in specific [59, 77]. However, to the best of our knowledge, these new paradigms and tools have yet to be applied to recommender systems. In this work, we address this knowledge gap and propose an initial test suite to evaluate the correctness of recommendation algorithms. This test suite contains both black box and white box tests at every level of abstraction, i.e., system, integration and unit. Along with this framework, we release `RecPack Tests`, an open-source Python package that contains test cases which researchers can use to test their own and others' recommendation algorithms and correct the issues that can affect their performance. We apply `RecPack Tests` to four open-source recommender systems packages, `Surprise` [32], `PyLensKit` [17], `RecPack` [47] and `Cornac` [63]. Despite the high test coverage of each of these packages, we find that we are still able to uncover bugs, undocumented functional requirements and interesting undesirable behaviors of algorithms. This validates our thesis that testing provides a valuable additional perspective on the evaluation of recommendation algorithms. Without testing our recommendation algorithms, we run the risk of drawing conclusions based on incorrect results, which is detrimental to progress in scientific research on recommender systems.

2 BACKGROUND

Currently, traditional software systems are tested, whereas recommender systems and other machine learning systems are most often said to be evaluated. In this Section, we explore in which ways testing and evaluation are different, and how testing can provide a complementary perspective when evaluating machine learning systems, and recommender systems in particular. First, in Section 2.1 we lay the groundwork for our discussion and discuss the main concepts and terms we will use throughout this work. In Section 2.2, we describe how machine learning systems are typically evaluated and look at the practice of evaluating machine learning systems from the perspective of software testing. In Section 2.3 we continue with a brief discussion of how recommender systems are evaluated and how testing could complement their evaluation. Finally, in Section 2.4, we examine recent work on testing machine learning systems.

2.1 Testing Software Systems

In the broadest sense, software testing encompasses any action aimed at discovering software failures so that they may be corrected. These actions can range from code review to unit testing or

even online AB testing. However, in the remainder of this work we will use the narrower definition, as proposed by ISO/IEC: "software testing is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [1].

In automated testing, we typically define a 'test suite' consisting of a series of 'test cases'. A test case is "a set of test inputs, execution conditions and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement" [1]. Further, a 'positive test case' is a valid test input for which we expect our software system to provide the correct result. If an error is encountered, or the result is not as expected, the positive test case will fail. A 'negative test case', on the other hand, is an invalid input test for which we expect our software system to fail gracefully, for example, by displaying an appropriate error message. Each test case tests some 'test criteria'. These test criteria are the "criteria that a system or component must meet in order to pass a given test" [1]. Finally, the 'test inputs', sometimes referred to as 'test data' or 'test fixtures', are the "data created or selected to satisfy the input requirements for executing one or more test cases" [1].

Software tests are typically run before the software system is deployed. The behavior of traditional software systems is deterministic, and therefore it is reasonable to assume that a program will not suddenly fail a test case it succeeded previously. After deployment, monitoring and logging are used to uncover bugs that were not spotted during testing. When such a bug is discovered a test case is written that reproduces it, after which the behavior is corrected and a new version is deployed.

In the remainder of this Section, we limit ourselves to a discussion of those concepts and terms that will support the later discussion of testing machine learning and recommender systems. For a more complete overview of software testing, we refer the interested reader to the seminal book on the subject, 'The Art of Software Testing' [50].

2.1.1 Unit, Integration and System Level Testing. We can distinguish between different levels of abstraction at which a test can operate. Conventionally, the literature distinguishes between three levels: the system, integration and unit level [50]. However, the boundaries between these three levels of abstraction are somewhat blurred. The unit level is always the lowest possible level. The 'single responsibility principle' dictates that a function should fulfill a single purpose. A unit test, or test at the unit level, verifies whether such a function has fulfilled its purpose. Typically, a unit test case will first create a minimal, but representative, input to the function, then it will call the function, and finally it will compare the output of the function to the expected, correct, output. Take for example a function that sorts a list in ascending order: A test input could be the unsorted list `[2, 3, 1]`, whereas the expected output would be the same list but sorted `[1, 2, 3]`.

Testing at the unit level alone, even if we would theoretically be able to test all functions for all possible input values, is insufficient to catch every error in a software system. This is because software systems require different functions to work together to achieve some more complex task. For example, imagine we want to know the greatest possible difference between any two items in a list. This could be accomplished by first sorting the list in ascending order, after which the first value is subtracted from the last value in the sorted list. However, if the first function sorts the list in ascending order, while the second assumes the list was ordered in descending order, the output will be incorrect. Such tests of how different functions inter-operate, are typically called 'integration tests'. Integration tests can usually only be written after at least some of the individual components that make up a system have been implemented (and possibly verified by means of a unit test). System level testing then tests software systems as a whole after development has been completed. While unit and integration tests can verify that what is implemented was correctly

implemented, system level tests go beyond that and test whether or not the software system as a whole achieves its design goal.

2.1.2 Black or White Box Testing. The most important aspect of testing is the creation and design of effective test cases [51]. Testing a system 'completely' is impossible given a limited budget and limited amount of time. Therefore, a good testing strategy can help uncover the existence of bugs, but not prove that a program is entirely bug-free [50]. The question then becomes: "What subset of all possible test cases has the highest probability of detecting the most errors?" [51]

We can differentiate between two different perspectives when designing tests: black box and white box testing. Either test perspective can be used at any of the levels of abstraction discussed previously.

Black box testing is also known as data-driven, input/output-driven, behavioral or functional testing. As the name suggest, in black box testing we treat the component to be tested as a black box: We assume only the functional requirements of the component are known, not the actual implementation logic or source code. As a result, a black box test can verify if a program can accomplish what it is supposed to do provided the test inputs, but not give insight into how it accomplishes that goal. The component under test can be a function, in which case all we know about it is the function's signature: Its name, the inputs it takes and the values it returns. The component under test can also be the system as a whole, in which case we evaluate it as would an end user. Several strategies exist to determine how to achieve the highest possible confidence in the system with the least amount of test cases. A first important strategy to determine how many tests are required is 'equivalence class partitioning' [51]. In equivalence class partitioning, the idea is that a tester uses their knowledge of what input values can be considered realistic, and then divides the input space into classes of 'equivalent' values, i.e., groups of input values for which it is reasonable to assume the system will behave in the same way. For each representative class then, only one test case should be generated. Another strategy is 'boundary value testing'. The idea here is that test cases that explore the boundaries of the input domain are more difficult for the system to process than other values [51]. For example, a function that determines the sign of an input value may have no difficulty determining the sign of 100_000, but may struggle to determine the sign of $0.01e^{-16}$. Equivalence class partitioning and boundary value analysis can also be combined: The input space can be divided into a set of representative classes, and then the bounds of those classes are used to generate test cases. Another strategy is 'cause-effect graphing'. Here, the causes and effects are determined from the program specification and then used to generate test cases of the form: 'If this .. then that'. Typically, a formal Boolean logic network is created from the program specification [51]. A final strategy is 'error guessing': Given a particular program, using both intuition and prior experience to determine what are mistakes commonly made or what aspects of the functional specification are at risk of being overlooked [51]. Although hardly a formal approach, error guessing is found to be very useful in practice [51].

White box or glass box testing, on the other hand, designs tests on the basis of the implementation logic. Because of this, white box testing can uncover code faults that black box testing may have left undiscovered. However, as it starts from the implementation, not the functional requirements of the program, it cannot be used to test whether the program achieves its design goal. Instead, it can only verify that the program does what it does correctly. In white box testing, we know exactly what statements or branches have been executed and which have not. As a result, test completeness is most often evaluated in terms of test coverage. Test coverage is defined as the "degree, expressed as a percentage, to which specified test coverage items have been exercised by a test case or test cases" [1]. A first coverage criterion is 'statement coverage', i.e., every statement in the program is executed at least once [51]. Statement coverage can be used to identify unreachable

code, also known as dead code. However, it cannot guarantee that a program will always return the correct result. Therefore, it is considered a weaker criterion than 'branch coverage', also known as 'decision coverage'. This coverage criterion requires that at least every decision is taken once, i.e., every branch in the code is executed once [51]. This means that for every decision, i.e., if-else statement, in the code, we should make sure that the decision evaluates to both `True` and `False` at least once. An even stronger criterion then is 'decision/condition coverage', which requires that not only is every decision taken once, but also that every condition in a decision takes every possible outcome at least once [51]. Decision/condition coverage is therefore mostly useful when decisions contain multiple conditions, for example, if A > 3 and B < 4. With all of the coverage criteria, a tester needs to determine what percentage of coverage is required for them to have sufficient trust that the system is free of bugs.

Conventional wisdom dictates that a reasonable testing strategy contains a mix of both black box and white box tests [51]. Preference should first be given to the development of black box tests, which should then be supplemented with white box tests until the tester is reasonably confident that their program is free of bugs [51]. Both for white box and black box tests, both positive and negative test cases should be generated.

2.1.3 The Oracle Problem. In the previous Section, we discussed how to choose test data, i.e., the test inputs, to obtain the greatest possible trust in the system with the least amount of tests. However, a test case does not only require appropriate test inputs. We also need to know the desired outcome of the component under test for that input, so that we can evaluate whether or not the system behaved correctly [4]. To distinguish between correct and incorrect behaviors, we use a 'test oracle'. Defining such test oracles is not always easy, and therefore often referred to as the 'the oracle problem' [4, 46, 64]. Barr et al. [4] identify four different types of oracles, two of which are of interest to us: Specified and derived oracles.

We say an oracle is specified when the desired behavior of the component is fully specified and thus known. In many practical settings, this is not a realistic requirement: The desired behavior of a system given a test input may simply be too complex to express, or the program may have been written to find the answer to a question for which the answer was previously unknown.

When there is no full specification of the expected behavior of a system, one can try to define a 'derived oracle' instead. A derived oracle, also known as a pseudo-oracle or partial oracle, is a partial specification of the outputs.

In property-based testing, instead of verifying that the true result of a function matches the required output, we instead verify if a property of the desired output is upheld [39]. For example, whereas a test with a fully specified oracle would test if the list `[2, 3, 1]` has been transformed into `[1, 2, 3]` after sorting, a property-based test would instead check that for any input, the property of monotonicity, i.e., that every next item in a list should always be of the same or greater value that the previous, is upheld. We can expect this property to be true for all lists sorted in ascending order. Property-based tests avoid some of the difficulties associated with defining specified test oracles. If such properties exist, the derived oracles are often easier to express than the specified oracles. Unfortunately, such properties do not always exist.

Another important approach that relies on a form of derived oracle is metamorphic testing [11, 12, 64]. Rather than relying on an individual output to define an oracle, metamorphic tests use relationships between different input-output pairs to determine the correctness of the result of a component [64]. For example, to test if a modulo operation was correctly implemented, we could check that $\forall\, x, y \in \mathbb{N} : x \bmod y \equiv x + y \bmod y$, using a few representative values of x and y. The above example tests an invariance metamorphic relationship: The result should not change from one test input x to another $x + y$. Many other types of metamorphic relationships can be

expressed [64]. A disadvantage of metamorphic testing is that, like other types of property-based tests, it requires knowledge of the problem domain to express these metamorphic relations [64].

Finally, we want to highlight 'differential testing'. In differential testing, we assume that we have two comparable systems available [46]. In some cases, we can assume that one of the two systems will give us correct results. For example, we could want to try a faster or more memory-efficient implementation of a sorting function. Here, we could compare the results of our new function to the results of the previous function for a random set of inputs, to make sure that the results are still correct. In other cases, we cannot be sure of the correctness of either system. In these cases, we say that there is a bug in either one of the systems if they each give different results. Which system made the correct determination and which did not, is a question left to the tester.

2.2 Evaluating Machine Learning Systems

Machine learning systems are fundamentally different from traditional, deterministic software systems. Whereas in the latter case we program the exact behavior we expect the program to exhibit, in the former case this behavior is learnt from data. While virtually all traditional software systems are tested, machine learning systems are usually said to be 'evaluated'. In this Section we discuss the differences and commonalities between evaluating and testing machine learning systems.

Although many different applications of machine learning exist, some observations can be made regarding the evaluation procedures used to evaluate supervised learning approaches, which share some commonalities with the offline evaluation of recommender systems, in general. In supervised learning, a dataset is typically divided into a training, validation and test dataset. As suggested by the names, a training dataset is used to train the model, after which a validation dataset is used to determine the hyperparameters that correspond to optimal performance, until finally the performance on a test dataset can be obtained. For a typical classification task, performance is most often evaluated in terms of accuracy, precision or recall.

If we look at this evaluation procedure through a software testing lens, we spot many similarities. A test dataset is a set of previously unseen data points in the shape of input-output pairs, where the input consist of all features needed to create a prediction and the output is the expected label.

In essence, this is a form of black box testing where each data point is actually a test case for our system: The input features are nothing less than the test inputs and the labels are the test oracles. If we adopt this mindset, an important shortcoming of this evaluation procedure becomes obvious: Whereas software testers spend the majority of their time selecting the *right* test cases, in machine learning evaluation, we simply take these test cases from the data. However, there is no guarantee that the test cases in the test dataset are free of unwanted biases, or span the entirety of the allowed input space [70]. A systematic and automatic exploration of the allowed (and disallowed) input space, however, is crucial to build trust in the decisions machine learning systems make [15, 71]. Failing to systematically test machine learning models may lead to catastrophic error in production settings, especially in applications where safety is critical [70].

2.3 Evaluating Recommender Systems

Zangerle and Bauer [76] distinguish between three types of experiments that can be used to evaluate recommender systems: offline evaluations, user studies and online evaluations. Offline evaluations use a dataset of users' explicit or implicit feedback on items collected from a live system over a period of time [9]. This dataset is split into a training, validation and test dataset, according to some procedure, e.g., leave-last-one-out, and then the performance of different algorithms on one or more such datasets is compared [76]. Often, 'benchmark datasets' are used, which were made public for the purpose of repeated experimentation [9], e.g., the MovieLens datasets [28].

In user studies, a small set of human test subjects is recruited to perform some predefined task that requires them to interact with the recommender system, after which their experience with the system is evaluated using a combination of objective and subjective measures [76].

Finally, in online evaluations, the recommender system is deployed in a real-world, live setting, where users interact with the system organically to perform self-selected real-world tasks [76].

While online evaluations offer the greatest degree of 'reality', offline evaluations are by far the most common experiment type encountered in the academic literature [6, 76], often the only experiment type available to researchers in academia and are conducted by practitioners in industry as well to evaluate and select new ideas to evaluate online [9]. In the remainder of this Section, we will focus our attention on offline evaluation.

Recommendation algorithms are most often evaluated in terms of accuracy, although beyond-accuracy objectives are gaining popularity [76]. Current methods to evaluate the accuracy, also termed utility, of recommender systems in offline evaluations find their origins in methods used to evaluate other types of machine learning systems. The earliest works on recommender systems evaluated the systems with a focus on prediction error, e.g., by means of RMSE, accuracy or precision, similar to how supervised learning approaches are typically evaluated [9]. Later methods drew inspiration from the evaluation of information retrieval tasks and adopted metrics such as Precision@K, Recall@K or nDCG@K [9]. Although prediction error is still used by some, the metrics that were adopted from the domain of informational retrieval dominate the space today.

However, the offline evaluation of recommendation algorithms poses some additional difficulties that the evaluation of information retrieval systems does not. Most prominently, whereas in supervised learning and information retrieval the offline datasets used offer a reasonable approximation of the ground truth, in recommender systems, they often do not [9]. For example, in offline evaluation of information retrieval systems judgments are 'pooled' between different annotators to obtain ground truth datasets. On the other hand, offline datasets for the evaluation of recommendation algorithms typically consist of logged user interactions collected from a live system over some period of time. These datasets therefore suffer from significant presentation and selection biases [9]. In the context of recommender systems, 'correctness' is also a more elusive notion [9]. Whether a recommendation is accepted, e.g., clicked or bought, is dependent on the user's needs in that moment and other contextual factors, which are most often unobserved. This has fueled criticism that results obtained in offline evaluations have poor correspondence with online results [5, 6] and that we should look at beyond-accuracy objectives, and different experiment types, to get a better idea of how recommendation algorithms will perform in the real world [34, 76].

Offline evaluations have also received criticism for their apparent sensitivity to the exact experimental conditions. Recent independent reproducibility studies of 'state-of-the-art' recommendation algorithms have found results "don't add up", i.e., many methods published years ago are still able to outperform recent models [3, 18, 19, 58]. Even between these independent reproducibility studies, there is considerable disagreement on the performance ranking of different recommendation algorithms. For example, Anelli et al. [3] and Rendle et al. [58], although published within months of each other, do not agree on the ranking of algorithms, even for the same dataset and accuracy metric, as shown in Table 1. Also of note is that, despite only small differences between the data splits used in both works, the reported accuracy values differ greatly. This indicates that measuring performance in terms of accuracy measures in offline evaluation is not robust.

Finally we want to highlight that the relentless focus on 'accurate' recommender systems ignores important factors that contribute to the trustworthiness of recommender systems, such as transparency, fairness and robustness to noise and attack [74].

Because offline evaluation of recommendation algorithms is so prevalent, the community dedicates significant efforts towards proposing new evaluation metrics [e.g. 73] or data splits [e.g.

	Anelli et al. [3]	Rendle et al. [58]
EASE	0.336	0.449
SLIM	0.335	0.447
iALS	0.306	0.453
NeuMF	0.277	0.477

Table 1. NDCG@10 for the ML1M dataset for all algorithms that were evaluated in both Anelli et al. [3] and Rendle et al. [58]. Results taken directly from the respective papers.

35, 72] that do not suffer from the same shortcomings as their predecessors. However, the practice of evaluating a recommendation algorithm only by comparing its performance to that of other recommendation algorithms, using some evaluation metric that reports a single aggregate value across users, on a held-out dataset, is a constant throughout all works on offline evaluation. We argue that testing could provide an interesting complementary perspective on the (offline) evaluation of recommendation algorithms, in much the same way as it has for other machine learning systems. First, testing can ensure that a recommendation algorithm is reliable and free of errors, before further evaluations are performed. Second, the definition of carefully selected test cases would allow us to evaluate important properties such as the algorithm's robustness to noise and attack or undesired biases the model may have learned.

2.4 Testing Machine Learning Systems

Testing can provide a complementary perspective on the evaluation of machine learning systems, including recommender systems, that can increase our level of trust in the system's proper functioning. For the testing of machine learning systems, we can take inspiration from testing practices applied to traditional software systems [77]. However, the behavior of machine learning systems is dependent on both the code and data, which causes a number of additional challenges [70, 77].

Testing machine learning systems is a relatively new research domain. Interest in the domain has been steadily rising since 2016 [77] and continues to rise with the increased adoption of machine learning systems in practical settings [60], which has led researchers and practitioners to re-examine their reliability [7, 15, 68, 71, 77].

Naturally, application domains in which safety is critical, such as autonomous driving, have received the most attention [77]. Despite its relative novelty as a research domain, a number of success stories have been reported. For example, DeepXplore [55], a differential white box testing technique for deep learning models, was able to uncover thousands of incorrect behaviors in autonomous driving systems. OGMA [70], a differential black box technique for natural language processing classifiers, was similarly able to uncover thousands of erroneous behaviors.

There are four main difficulties when testing machine learning systems that we wish to highlight. In the remainder of this Section, we discuss these difficulties and how they have been addressed in the literature.

As mentioned earlier, a first issue is that the behavior of machine learning systems depends not only on the code, but also on the data [77]. This has two important consequences. Firstly, the system's behavior is no longer deterministic. As a result, the system may have learned the desired behavior from the data, but may also have picked up undesirable behaviors. This gives rise to several different 'properties' of machine learning systems that can be tested. In their survey, Zhang et al. [77] identify eight that have been of interest to the community: correctness, robustness, privacy, security, fairness, efficiency and interpretability. However, they stress that most papers they

collected focus on correctness and robustness. They define 'correctness' as "the ability of a machine learning system to get things right". 'Robustness' then, is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions". This interest in robustness is no surprise, as machine learning systems have been found to be susceptible to adversarial attacks that may exploit security vulnerabilities by using inputs that force models to produce erroneous outputs [10, 45]. Secondly, because the behavior is a function of the data, the behavior can also change every time new data is incorporated [77]. Therefore, unlike traditional software systems, machine learning systems should ideally be (re-)tested every time new data is incorporated [77].

A second important challenge is that the behavior of a machine learning system is often the result of different smaller components working together. This naturally migrates testing away from the unit level to the integration and system level [7, 59, 77]. Because of this, Riccio et al. [60] define the concepts of unit, integration and system level in machine learning systems as follows: They argue that the machine learning model should be considered a 'unit', and the parts surrounding it, such as mapping services and API, form the integration and system levels together with the machine learning model. However, this view assumes that the implementations of machine learning *algorithms* are already free of errors, which other works have shown to be incorrect [e.g. 55, 70].

Thirdly, machine learning systems suffer an especially difficult case of the oracle problem [36]. Machine learning systems are often used to answer questions for which no answers are known [70, 77], making it difficult to define specified oracles. Because of this, most of the literature on machine learning testing focuses on metamorphic and differential testing. Differential testing is of interest because it does not require the correct result to be known, but instead relies on disagreement between two very similar systems. As Udeshi and Chattopadhyay [70] write: "According to the robustness property, the classification classes of two similar inputs do not vary substantially for well trained machine-learning models". In other words, disagreement can be interpreted to mean one of both, or potentially both, systems are wrong.

Similarly, metamorphic testing is of interest because it relies only on metamorphic relationships between input-output pairs. Examples of the use of metamorphic tests in the natural language processing domain are found in Ribeiro et al. [59]. They propose a series of test cases that require either invariance to certain perturbations, or require that the label assigned to an output changes in a certain direction after perturbation. For example, they propose a metamorphic test that requires that if a neutral word is replaced with another neutral word, the sentiment assigned to a sentence does not change. A disadvantage of metamorphic testing is that it relies heavily on domain-specific knowledge, and thus requires the definition of reasonable metamorphic relations for every problem domain individually [70]. Another important type of test that has been designed specifically for machine learning systems is what we will call a 'minimal functionality test', as in Ribeiro et al. [59]. A similar idea was also described in Breck et al. [7] where it was referred to as 'purposeful overfitting'. The idea of both types of tests is to validate that the machine learning model is learning reliably. In the former, the 'minimal functionality' required is that negation changes the sentiment of a sentence. In the latter, they try to get a model to overfit the training data. If it is unable to do so, this points to an error in the algorithm's implementation. They also mention the use of 'toy examples', i.e., small datasets with straightforward patterns, to validate algorithm implementations. The purpose is then to learn the pattern present in the toy example, which can be validated by means of a specified oracle. However, determining suitable error tolerances on these specified oracles poses another problem [36].

A final challenge is to determine when a machine learning system has been tested sufficiently [36]. Machine learning systems have extremely large and complex input spaces in comparison to traditional software systems. This makes applying techniques such as equivalence class partitioning and

boundary value analysis, to reduce the number of test cases required, infeasible. Manually selecting a sufficient set of test cases is similarly challenging. Additionally, training machine learning systems often takes a long time, so running a large number of test cases to satisfy coverage criteria may also be impractical. Some efforts have been made to define new completeness criteria and strategies for test case generation, specific to machine learning systems. For example, Sun et al. [68] suggest four coverage criteria for white box testing of deep neural networks. They highlight that simple coverage criteria, such as 100% neuron coverage, are trivial to achieve and may lead to insufficient testing. Udeshi and Chattopadhyay [70] then propose a grammar-based test case generation strategy for natural language processing classifiers.

Finally, we note that within the domain of recommender systems, prior work on testing is limited. In their abstract, Saberian and Basilico [61] write that unit and integration tests are an important aspect of 'RecSysOps', but do not describe the actual tests used. Then, with RecList, Chia et al. [13] propose a framework to facilitate behavioral (black box) testing of recommender systems. However, their focus is on providing the necessary tooling to allow easier definition of tests, not providing a test suite of predefined test cases. The test cases they do propose are not automated tests that use an oracle to determine success or failure, but rely on human judgment of the outputs, e.g., plots, instead. Finally, there exists some work that proposes methods to verify the robustness of the system under attack, usually shilling or profile injection attacks [e.g. 26, 49]. Such an attack can be seen as a type of test: A test input is generated that contains some amount of false profiles, after which the behavior of the new model can be compared to the previous, clean, model. However, again, currently these tests are not formulated as an automated test suite.

3 TESTING THE CORRECTNESS OF RECOMMENDATION ALGORITHMS

In this work, we propose an initial test suite to evaluate the correctness of implicit feedback, collaborative filtering recommendation algorithms, i.e., their ability to 'get things right' [77]. To define the test cases, we use a combination of 'error guessing', drawing from our experience building `RecPack`, as well as draw inspiration from prior work on testing machine learning systems, discussed in Section 2.4. It is important to highlight that our test suite does not come with any guarantees of completeness. Instead, it is intended as a first step towards comprehensive testing of recommendation algorithms.

The test suite comprises a combination of black box and white box tests, at every level of abstraction. For the purpose of this work, we define the unit, integration and system levels as follows. We use the term 'system' to refer to a recommendation algorithm trained on real data, i.e., a recommendation model. We believe this definition of the term is most similar to how it is used in software testing, where system level tests are used to validate whether the system as a whole achieves its design goal. Similarly, the system level tests we propose here validate whether our recommendation models are able to achieve their design goals. In our 'integration' tests, we aim to test the correctness of the implementation of the recommendation algorithm. Like other machine learning algorithms, recommendation algorithms often consist of several components that need to work together. Our integration tests validate whether they do so correctly. Finally, in our 'unit' tests, we validate that specific functions or small components of the algorithm function as expected. As argued in Section 2.4, the behavior of any machine learning system or model is usually the result of different smaller components working together, and therefore most of the tests we propose here are situated at the integration level.

Even within the subdomain of implicit feedback, collaborative filtering recommendation algorithms, there is great diversity in algorithms. They range from complex neural networks to models that have simple, closed-form solutions, such as EASE^r [66]. Because a thorough discussion of this entire 'algorithm design space' would lead us too far, we focus on a few properties of algorithms

that are shared between many collaborative filtering recommendation algorithms. A first property is the task the recommendation algorithm is trained and evaluated on. The two most common tasks are top-K recommendation and sequence-aware (sequential) recommendation [40, 56]. An implicit feedback dataset typically consists of logged interactions of users with items at some time in the past. In top-K recommendation, we make the implicit assumption that every one of these logged interactions is equally valuable to predict the top-K items the user is most likely to interact with next. On the other hand, in sequence-aware recommendation it is assumed that the order of interactions matters, and a user's most recent interactions are more predictive of what they will do next [40, 56]. We will distinguish between these two tasks in our black box tests when appropriate.

A second property is the representation of items and/or users the recommendation algorithm learns. Once again, we focus on the two most common types of representations learned by collaborative filtering recommendation algorithms. The first representation is a matrix of similarities between items. This representation is used by all item neighborhood-based algorithms, but many others too, e.g., EASE^r[66]. The second representation is an item embedding in some latent space. This representation is used by all matrix factorization algorithms and many deep learning algorithms, e.g., most graph neural networks. We will distinguish between these two representations in our white box tests, where we assume access to the implementation internals.

Finally, we note that recommendation models can be learned in different ways. Some algorithms, such as EASE^r [66] and ItemKNN [16] have a closed-form solution. Others learn a model by means of alternating least squares (ALS) or stochastic gradient descent, which require several iterations to obtain a final solution. Typically, these 'iterative' methods require initial values from which to start learning increasingly better solutions. We will propose some tests, both black and white box, that are only applicable to iterative algorithms.

We will assume that every collaborative filtering recommendation algorithm implements two methods: one to fit a model and one to obtain predictions from a model. This assumption is generally true, at least in publicly available implementations [such as 2, 17, 32, 47].

In Table 2 we give an overview of the tests we propose, the questions they answer, and the types of collaborative filtering recommendation algorithms they are applicable to. The remainder of this Section is structured as follows. First, we introduce some notation that will be used throughout this Section in the presentation of test cases. Then, tests are organized by level of abstractions and whether they rely only on functional requirements (black box) or require access to implementation details (white box).

Notation

We will use $U = \{u_1, u_2, \ldots\}$ to denote a set of users and $I = \{i_1, i_2, \ldots\}$ to denote a set of items. We use X to denote any matrix of size $|U| \times |I|$.

We use $X^{\text{train}} \in \{0,1\}^{|U|\times|I|}$ to denote the training dataset used to fit a Top-K recommendation model and $X^{\text{test}} \in \{0,1\}^{|U|\times|I|}$ the test dataset, i.e., the user-item interaction matrix used as input at prediction time. Finally, we will use $X^{\text{out}} \in \{0,1\}^{|U|\times|I|}$ to denote the matrix of interactions we are trying to predict. We will use X_u to refer to the row vector of X corresponding to a user u. Individual elements of a matrix are denoted using lowercase letters, e.g., the value of x_{ui}^{train} indicates whether a user u has an interaction with item i in the training dataset.

For the sequential recommendation task, we will represent the training and test dataset as a set of sequences $\mathcal{S} = \bigcup_{u\in U} S_u$ where every $S_u = (s_0, s_1, \ldots, s_L)$ can be of arbitrary length L and every element s_k is an item from I the user has interacted with. Then, $\mathcal{S}^{\text{train}}$ is used to denote the training dataset and $\mathcal{S}^{\text{test}}$ the test dataset. Typically, the goal in sequential recommendation is to predict $S^{\text{out}} = (s_k, \ldots, s_L)$ based on $S^{\text{test}} = (s_0, \ldots, s_{k-1})$.

Test ID	Description	Task	Item Representation	Learning Method
BBST1+	Does the model make recommendations for the right users?	-	-	-
BBST1-		-	-	-
BBST2	Does the model recommend items a user previously interacted with?	-	-	-
BBST2^c		-	-	-
BBST3	Is the algorithm prone to sub-optimal local optima?	-	-	Iterative
WBST1Sim+	Does the algorithm learn representations for the right items?	-	Similarities	-
WBST1Sim-		-	Similarities	-
WBST1Emb+		-	Embeddings	-
WBST1Emb-		-	Embeddings	-
BBIT1TopK	Is the algorithm able to learn simple patterns in the training data?	Top-K	-	-
BBIT1Seq		Seq.	-	-
BBIT2TopK-A	Does the algorithm effectively personalize recommendations?	Top-K	-	-
BBIT2TopK-B		Top-K	-	-
BBIT2Seq-A		Seq.	-	-
BBIT2Seq-B		Seq.	-	-
BBIT3	Can the algorithm use new interactions at prediction time?	-	-	-
BBIT4		-	-	-
BBIT5	Can results be reproduced?	-	-	-
BBIT6		-	-	-
WBIT1Emb	Is the algorithm able to learn simple patterns in the training data?	Top-K	Embeddings	-
WBIT1Sim		Top-K	Similarities	-
WBIT2Sim	Does the algorithm learn self-similarity?	Top-K	Similarities	-
WBUT1Iter+	Are gradients updated at the right time?	-	-	Iterative
WBUT1Iter-		-	-	Iterative

Table 2. Overview of the test suite. Every test case is assigned a Test ID which identifies a test case by its attributes. Its type: black (BB) or white box (WB), system (ST), integration (IT) or unit (UT); positive (+) and negative (-) test cases; complementary test cases (^{c}); their defining property, e.g., sequential recommendation (Seq); and finally, multiple tests with a different approach that test the same property (-A, -B). For every test case we provide a description and the task, item representation and learning method they can be used for. A '-' indicates the test can be applied to all collaborative filtering recommendation algorithms regardless of the task, item representation or learning method.

Finally, we will use M to denote a fitted recommendation model and X^{pred} to denote the predictions obtained from the model, $X^{\text{pred}} = M(X)$ for the top-K recommendation task and $X^{\text{pred}} = M(S)$ for the sequential recommendation task.

3.1 Black Box System Tests

We propose a set of black box system tests that verify fundamental properties of a recommendation model. The first two tests, *BBST1* and *BBST2* are applicable to all collaborative filtering recommendation algorithms, whereas *BBST3* is only applicable to iterative algorithms. If the test cases succeed, the tester can be confident that the performance metrics they compute are not unduly influenced by the random seed used to compute the model, or users for whom the model cannot make reasonable recommendations, and that the model exhibits expected behavior with regards to re-recommendation, i.e., recommendation of items the user interacted with previously.

3.1.1 BBST1: Does the model make recommendations for the right users? In offline evaluation, as detailed in Section 2.3, we typically compute one or more metrics to evaluate the performance of our recommendation algorithms. Most of these metrics, for example, NDCG@K and Recall@K, will compute a score for each ranked top-K list of recommendations presented to a user and then take the average across all users to be the final score for this recommendation algorithm. Thus, if we fail to make recommendations for a user for whom we expect to make recommendations our evaluation metrics will be affected negatively. Alternatively, trying to make recommendations for a user for whom we do not have any information may lead to recommendations of low quality. Below, we propose two test cases that when used together verify that the algorithm makes recommendations for the 'right' users only.

BBST1+: Assert all 'known' users receive recommendations. Different collaborative filtering algorithms may have different interpretations of what are 'known' users. A first group of 'item-based algorithms' learn only a representation of the items and treat users as a bag or sequence of items at prediction time. These algorithms can make reasonable recommendations for all users u with interactions in X^{test}, regardless of whether they had any interactions in X^{train}. In contrast, user-based algorithms, which learn a representation of the user at fitting time, require that a user u had interactions in X^{train}. To accommodate for both, we set $X^{\text{test}} = X^{\text{train}}$. Now, we train a model M on the training dataset and generate predictions $X^{\text{pred}} = M(X^{\text{test}})$. We then create vector $Y \in \{0,1\}^{|U|}$ depicting the users for which the model made predictions, where element

$$y_u = \begin{cases} 1 & \text{if } \exists\, i : x_{ui}^{\text{pred}} > 0 \\ 0 & \text{otherwise} \end{cases}$$

and partial oracle $Z \in \{0,1\}^{|U|}$ depicting the users for which we expected the model to make predictions, where element

$$z_u = \begin{cases} 1 & \text{if } \exists\, i : x_{ui}^{\text{test}} > 0 \\ 0 & \text{otherwise.} \end{cases}$$

We require that

$$Z \cdot Y = \sum_{u \in U} z_u$$

i.e., the model made predictions for all users we expected it to make predictions for.

BBST1-: Assert no 'unknown' user receives recommendations. Now, we define the negative test case that is the counterpart to *BBST1+*. Here, we verify that 'unknown' users do not receive recommendations. Again, we take $X^{\text{test}} = X^{\text{train}}$ and train a model M which we then use to generate predictions X^{pred}. Using Z and Y as defined in the previous test, we now require that the model made no predictions for users we did not expect it to make predictions for. Formally,

$$\neg Z \cdot Y = 0.$$

3.1.2 BBST2: Does the model recommend items a user previously interacted with? In some use cases, it can be beneficial to have an algorithm make recommendations of items the user has interacted with before.

For example, in music recommendation, users will listen to the same song or artist over and over again. In other use cases, it may be beneficial to never recommend an item again after an interaction has taken place, e.g., users will typically read the same news article just once. It is therefore important to verify whether the algorithm matches the functional requirement of the use case regarding re-recommendation. If the algorithm does re-recommend when it is not supposed to, it is essentially wasting valuable slots in the ranked top-K recommendation list on items it knows a user will not interact with again. This gives it a significant disadvantage compared to algorithms that do take this functional requirement into account. We propose two complementary test cases to verify that either the algorithm does, or does not re-recommend items a user has interacted with previously.

BBST2: Assert model recommends items a user previously interacted with. Using a property-based test, we verify that at least N users receive a recommendation of an item they previously interacted with in their ranked top-K list of recommendations, where both N and K can be configured by the tester. We assert that

$$|\{u : \text{top-K}(X_u^{\text{pred}}) \cap I_u^{\text{train}} \neq \emptyset\}| \geq N,$$

where the function top-K is defined as

$$\text{top-K}(X_u^{\text{pred}}) := \text{argmax}_{I' \subset I, |I'|=K} \sum_{i \in I'} x_{ui}^{\text{pred}} \tag{1}$$

and the set of items I_u as

$$I_u := \{i : x_{ui} > 0\} \tag{2}$$

for any X, e.g., X^{train} (I_u^{train}) and X^{test} (I_u^{test}).

BBST2^c: Assert model never recommends items a user previously interacted with. To accommodate use cases where it is undesirable that a model recommends items a user previously interacted with, we present a complementary test case to *BBST2*. With top-K and I_u as defined in Equations 1 and 2 respectively, we assert that

$$|\{u : \text{top-K}(X_u^{\text{pred}}) \cap I_u^{\text{train}} \neq \emptyset\}| = 0.$$

3.1.3 BBST3: Is the algorithm prone to sub-optimal local optima?

BBST3: Assert impact of changing the seed on performance is below threshold. In theory, an iterative algorithm should be able to obtain the same performance regardless of the random seed used to set the initial values, given sufficient training epochs and an appropriate learning rate. However, it is well-known that some neural architectures are more prone to getting stuck in sub-optimal local optima [67]. It is therefore desirable to verify how reliant a recommendation algorithm is on finding the 'right seed' to lead to good recommendation performance. The performance of recommendations is measured using a variety of metrics, as discussed in Section 2.2. Here, as an example, we will use the Recall@K metric, computed as

$$\text{Recall@K}(X^{\text{pred}}) := \frac{1}{|U|} \sum_{u \in U} \frac{\sum_{i \in \text{top-K}(X_u^{\text{pred}})} x_{ui}^{\text{out}}}{\sum_{j \in I} x_{uj}^{\text{out}}},$$

with top-K as defined in 1. In this test, we will train n recommendation models, $M_1, M_2, \ldots, M_n$, with n configurable, on the same training data, X^{train}, using the same hyperparameters but different random seeds. For each of the learned models, we then evaluate the performance in terms of Recall@K. We verify that the coefficient of variation of these n values of Recall@K, $CV = \frac{\sigma}{\mu}$, is below a threshold: $CV < \xi$. Parameters K, n and ξ should be defined by the tester. For ξ, we recommend values $0.1 < \xi < 0.25$.

3.2 White Box System Tests

The white box system tests we propose here verify an expected property of the internal representation of a trained recommendation model. For both algorithms that learn item similarities and those that learn item embeddings, we can expect that the model has learned something about every item in the training dataset. Conversely, we want the model to be aware that it has no knowledge of an item that was not in the training dataset and therefore make no recommendations based on these items. However, when an iterative algorithm sets initial values, it typically does so for all items, not just the items in the training dataset. As a result, items that were not in the training dataset will have a random representation. Therefore, for both item representations, we propose a positive and a negative test case that should be used in conjunction to verify that the algorithm indeed only makes recommendations for the right items.

3.2.1 WBST1Sim: Does the model learn item similarities for the right items? Item similarity models, such as item-item neighborhood models, learn similarities between the items in the training dataset at fitting time and then typically represent a user as a bag or sequence of items at prediction time. Therefore, the trained model should have learned neighbors for all items in the training dataset and have learned no neighbors for all items that were not a part of the training dataset.

WBST1Sim+: Assert all items in the training dataset have neighbors. In this positive test case we verify the property that every item in the training dataset has neighbors. Formally, we define the neighbors of item i as the set of items which optimises the following:

$$\text{neighbors}(i) := \text{argmax}_{I' \subset I} \sum_{j \in I'} \text{sim}(i, j) \tag{3}$$

where $\text{sim}(i, j)$ is defined as the similarity between item i and item j. Then we define the set of items in the training dataset as

$$I^{\text{train}} = \bigcup_{u \in U} I_u^{\text{train}}, \tag{4}$$

with I_u^{train} as defined in Equation 2. We then require that

$$\forall i \in I^{\text{train}} : \text{neighbors}(i) \neq \emptyset.$$

WBST1Sim-: Assert no item that was not in the training dataset has neighbors. In this negative test case we verify the property that all items that were not in the training dataset do not have neighbors. Formally,

$$\forall i \in I \setminus I^{\text{train}} : \text{neighbors}(i) = \emptyset$$

with neighbors(i) as defined in Equation 3.

3.2.2 WBST1Emb: Does the model learn embeddings for the right items? Some collaborative filtering algorithms, such as matrix factorization algorithms, learn embeddings of items, and sometimes also users, at fitting time. At prediction time, a user is either represented as a user embedding vector or a bag or sequence of items. Therefore, we require that the trained model has an embedding for

every item in the training dataset and sets the embedding of all items that were not in the training dataset to zero, such that they can have no influence on predictions.

WBST1Emb+: Assert all items that were in the training dataset have a nonzero embedding. In this positive test case, we verify that every item in the training dataset has a nonzero embedding and as such can contribute to recommendations. We will represent the item embeddings as $V \in \mathbb{R}^{|I| \times K}$, with K the dimension of the latent space, and use $V_i \in \mathbb{R}^K$ to indicate the item embedding vector of item i. Then, after a model M has been trained, we assert that

$$\forall i \in I^{\text{train}} : V_i \neq \mathbf{0}$$

with I^{train} as defined in Equation 4.

WBST1Emb-: Assert all items that were not in the training dataset have a zero embedding. In this negative test case, we verify that no item that was not in the training dataset has a nonzero embedding. This way, these items can never be recommended. Formally,

$$\forall i \in I \setminus I^{\text{train}} : V_i = \mathbf{0}.$$

3.3 Black Box Integration Tests

When a recommendation model fails to make good predictions, this can have many causes: the data it was trained on may not be representative or we may be using the wrong hyperparameters. However, recommendation algorithms can also fail due to flaws in their design or other implementation errors. Therefore, we propose a set of black box integration tests intended to verify that a recommendation algorithm's design is sound and free of errors. More specifically, we propose a series of minimum functionality tests to verify the algorithm's capacity to learn reasonable representations and actually personalize recommendations. Tests *BBIT1, BBIT2* and *BBIT3* verify common functional requirements of recommendation algorithms. Additionally, tests *BBIT4* and *BBIT5* verify if results are reproducible, which is an important requirement in the context of offline evaluation. All of the tests proposed are applicable to all algorithms regardless of representation or task.

3.3.1 BBIT1: Is the algorithm able to learn simple patterns in the training data? To verify whether the recommendation algorithm is able to learn meaningful patterns, we can perform a minimum functionality test. In such a test, the recommendation algorithm is given a very simple training dataset with a pattern that is easy to overfit. We then verify if the algorithm was indeed able to learn the pattern. The required minimum functionality depends on the recommendation task, and therefore we define separate tests for every task.

BBIT1TopK: Assert algorithm can overfit the training dataset in a Top-K recommendation task. The purpose of this test is to verify that an algorithm developed for the top-K recommendation task has the capacity to learn. To do so, we create a toy dataset $X^{\text{toy}} \in \{0,1\}^{|U| \times |I|}$ with $|U|$ and $|I|$ to be defined by the tester. We recommend relatively small values, e.g., $|U| = |I| = 100$. This dataset is populated as follows. Let $A, B \sim \text{Bernoulli}(\rho)$, with ρ a configurable parameter which represents the density of matrices $A, B \in \{0,1\}^{\frac{|U|}{2} \times \frac{|I|}{2}}$. We recommend setting $\rho > 0.3$. Then, we define

$$X^{\text{toy}} = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}.$$

We use U^A to denote the users belonging to A and U^B the users belonging to B. Due to the way we constructed X^{toy}, there is no overlap between both sets of users, i.e., $U^A \cap U^B = \emptyset$. The same property applies to the items: If we use I^A to denote the items belonging to A and I^B the items belonging to B, $I^A \cap I^B = \emptyset$. Because of this, we can expect a model M trained on X^{toy} to rank all

items in I^A higher than items in I^B for every user in U^A, and vice versa. Therefore, we will assert that

$$\forall u \in U^A, i \in I^A, j \in I^B : M(x_{ui}^{\text{toy}}) > M(x_{uj}^{\text{toy}})$$

and

$$\forall u \in U^B, i \in I^A, j \in I^B : M(x_{uj}^{\text{toy}}) > M(x_{ui}^{\text{toy}}).$$

BBIT1Seq: Assert that the algorithm can overfit a sequence of interactions in a sequential recommendation task. Similar to the previous test, we wish to verify if the recommendation algorithm has the capacity to learn, but then in the context of a sequential recommendation task. To do so, we create another toy dataset $\mathcal{S}^{\text{toy}}$, which is a set of ordered interaction logs. We construct $\mathcal{S}^{\text{toy}}$ such that it is made up of two sets of users, U^A and U^B, where $|U^A| = |U^B| = m$ is a parameter to be defined by the tester. For each of these sets of users we sample a proto-sequence of items $S^A, S^B \sim \text{Mult}(n, |I|, P_{|I|})$, where the parameter n is the length of the proto-sequences and every item i has equal probability of being sampled, i.e. $P_{|I|} \sim \text{Unif}(0, |I|-1)$. Now, we assign to every user $u \in U^A$ a sub-sequence of S^A and to every user $u \in U^B$ a sub-sequence of S^B . We use $S_{k:l}$ to denote the sub-sequence of a sequence S between positions k and l, where $0 \leq k < l$ and $l < n$. Formally, $\forall u \in U^A, \exists\, k, l : S_u^A = S_{k:l}^A$, and to every user $u \in U^B$ a sub-sequence of S^B, $\forall u \in U^B, \exists\, k, l : S_u^B = S_{k:l}^B$. We can then define our toy dataset as

$$\mathcal{S}^{\text{toy}} = (\bigcup_{u \in U^A} S_u^A) \cup (\bigcup_{u \in U^B} S_u^B).$$

We can now train a model M on $\mathcal{S}^{\text{toy}}$. Next, we evaluate its ability to predict element $j+1$ for every sub-sequence $s_{0:j}$ of both S^A and S^B. Formally, we assert that

$$\forall S_{0:j} \in S^A : \text{top-1}(M(S_{0:j})) = s_{j+1}$$

and

$$\forall S_{0:j} \in S^B : \text{top-1}(M(S_{0:j})) = s_{j+1}.$$

Depending on whether or not the recommendation algorithm allows re-recommendation of items that a user has interacted with in the past, S^A and S^B can be selected such that they either do or do not contain repetitions.

3.3.2 BBIT2: Does the algorithm effectively personalize recommendations? Most recommendation algorithms have the purpose of providing personalized recommendations to each individual user. In this minimum functionality test, we verify the algorithm's ability to do so by checking that the algorithm serves different recommendations to users with different interaction histories. Once again, we create separate test cases for Top-K recommendation and sequential recommendation, as each task makes different assumptions.

BBIT2TopK-A: Assert similar users receive different recommendations in a Top-K recommendation task. In this test, we wish to verify that similar users do not receive exactly the same recommendations. To do so, we create another toy dataset X^{toy}. First, we sample a user-item interaction matrix $A \in \{0,1\}^{|U| \times |I|} \sim \text{Bernoulli}(\rho)$, where ρ represents the density of A. With I_u as defined in Equation 2, we use I_u^A to denote the items that user $u \in U$ has interacted with according to A. Now, we create another matrix $\tilde{A} \in \{0,1\}^{|\tilde{U}| \times |I|}$ where $\tilde{U}$ is of the same cardinality as U, $|\tilde{U}| = |U|$. We require that the items each user $\tilde{u} \in \tilde{U}$ has interacted with are a subset of the items user $u \in U$ interacted with. Formally,

$$\forall u \in U, \exists\, \tilde{u} \in \tilde{U} : I_{\tilde{u}}^{\tilde{A}} \subset I_u^A.$$

We express the cardinality of $I_{\tilde{u}}^{\tilde{A}}$ as a percentage of the cardinality of I_u^A, i.e., $|I_{\tilde{u}}^{\tilde{A}}| = \psi |I_u^A|$. We recommend setting $0.7 < \psi < 1$. We assume the set of items $I_{\tilde{u}}^A$ is sampled uniformly at random from I_u^A. Now we have a new user $\tilde{u}$ that is very similar to user u for every user $u \in U$. We can formally define $\tilde{A}$ with elements $\tilde{a}_{\tilde{u}i}$ as

$$\tilde{a}_{\tilde{u}i} = \begin{cases} 1 & \text{if } i \in I_{\tilde{u}}^{\tilde{A}} \\ 0 & \text{otherwise.} \end{cases}$$

and

$$X^{\text{toy}} = \begin{bmatrix} A \\ \tilde{A} \end{bmatrix}.$$

Next, we train a model M on X^{toy} and request recommendations for this same dataset, $X^{\text{pred}} = M(X^{\text{toy}})$. We assert that for every pair of users u and $\tilde{u}$, their top-K recommendations are different, with K to be defined by the tester. Formally,

$$\forall (u, \tilde{u}) : \text{top-K}(X_u^{\text{pred}}) \neq \text{top-K}(X_{\tilde{u}}^{\text{pred}}),$$

with top-K as defined in Equation 1.

BBIT2TopK-B: Assert similar users receive different recommendations in a Top-K recommendation task. In this test, we offer a different approach to verifying that similar users do not receive exactly the same recommendations. Rather than removing some interactions of a user we replace some interactions with different items. With $I_{\tilde{u}}^{\tilde{A}}$ as defined above, we define

$$\hat{I}_{\tilde{u}} = I_{\tilde{u}}^{\tilde{A}} \cup I_u^{\text{rand}}$$

where

$$I_u^{\text{rand}} \sim \text{Mult}(n_u, |I|, P_{|I|})$$

with n_u the number of items sampled for user u and $P_{|I|}$ assigning uniform probability to every item. We take n_u such that $\forall (u, \tilde{u}) : n_u = |I_u| - |I_{\tilde{u}}|$. We then define $\hat{A}\{0,1\}^{|\tilde{U}| \times |I|}$, where each element $\hat{a}_{\tilde{u}i}$ is

$$\hat{a}_{\tilde{u}i} = \begin{cases} 1 & \text{if } i \in \hat{I}_{\tilde{u}} \\ 0 & \text{otherwise.} \end{cases}$$

and

$$X^{\text{toy}} = \begin{bmatrix} A \\ \hat{A} \end{bmatrix}.$$

Following the previous test, we again assert that the model M trained on X^{toy} gives different predictions for pairs of users u and $\tilde{u}$ Formally,

$$\forall (u, \tilde{u}) : \text{top-K}(X_u^{\text{pred}}) \neq \text{top-K}(X_{\tilde{u}}^{\text{pred}}).$$

BBIT2Seq-A: Assert similar users receive different recommendations in a sequential recommendation task. In this test, we verify that similar users do not receive the same recommendation in a sequential recommendation task. To do so, we create a toy dataset $\mathcal{S}^{\text{toy}}$. For this, we sample $|U|$ sequences $S_u \sim \text{Mult}(n_u, |I|, P_{|I|})$ where n_u is the length of the sequence, $|I|$ the total number of items in the toy dataset and every item i has equal probability of being sampled, i.e. $P_{|I|} \sim \text{Unif}(0, |I| - 1)$ and define the set $\mathcal{S}_U = \bigcup_{u \in U} S_u$. As in test case *BBIT2TopK-A*, we create a user $\tilde{u}$ that is similar to user u by taking a subset of that user's interaction history. Because in the sequential recommendation task later interactions of a user carry more weight when making recommendations, we create this

user $\tilde{u}$ such that they have the same interaction history up to $k-1$, with $k = \psi n_u$ and $0.7 < \psi < 1$. Formally,

$$\forall u, \exists\, \tilde{u} : S_{\tilde{u}} = S_{u_{0:k-1}}.$$

We can then define the set of sequences $\mathcal{S}_{\tilde{U}} = \bigcup_{\tilde{u}\in\tilde{U}} S_{\tilde{u}}$. Now, we train a model M on $\mathcal{S}^{\text{toy}} = \mathcal{S}_{\tilde{U}} \cup \mathcal{S}_U$ and request recommendations $X^{\text{pred}} = M(\mathcal{S}^{\text{toy}})$. Similar to the previous two test cases, we verify that for every pair of users u and $\tilde{u}$, their top-K recommendations are different. Formally,

$$\forall(u, \tilde{u}) : \text{top-K}(X^{\text{pred}}_u) \neq \text{top-K}(X^{\text{pred}}_{\tilde{u}}).$$

BBIT2Seq-B: Assert similar users receive different recommendations in a sequential recommendation task. In this test case, we offer a second approach to verify that similar users don't receive exactly the same recommendations for the sequential recommendation task, in a similar vein to *BBIT2TopK-A* and *BBIT2TopK-B*. As in *BBIT2TopK-B*, rather than removing some interactions of a user u to create similar user $\tilde{u}$, we swap these interactions for a randomly selected set of items. Formally,

$$\forall \tilde{u} : \hat{S}_{\tilde{u}} = (S_{\tilde{u}}, S^{\text{rand}}_u)$$

with $S^{\text{rand}}_u \sim \text{Mult}(n_u - k, |I|, P_{|I|})$ where n_u is the length of S_u. We then create $\hat{\mathcal{S}}_{\tilde{U}} = \bigcup_{\tilde{u}\in\tilde{U}} \hat{S}_{\tilde{u}}$. Again we train a model M on $\mathcal{S}^{\text{toy}} = \hat{\mathcal{S}}_{\tilde{U}} \cup \mathcal{S}_U$ and request recommendations $X^{\text{pred}} = M(\mathcal{S}^{\text{toy}})$. Once again, we verify that for every pair of users u and $\tilde{u}$, their top-K recommendations are different. Formally,

$$\forall(u, \tilde{u}) : \text{top-K}(X^{\text{pred}}_u) \neq \text{top-K}(X^{\text{pred}}_{\tilde{u}}).$$

3.3.3 BBIT3-4: Can the algorithm use new interactions at prediction time? In some offline evaluation scenarios, such as the 'strong generalization' scenario [47], users are split such that they only occur in either the training or the test dataset. In other offline evaluation scenarios users do occur in both datasets, but with different interaction histories [47]. To verify whether the recommendation algorithm is compatible with the desired offline evaluation scenario, we propose two tests that verify if the algorithm is able to make predictions for a new user (*BBIT3*) and can use new interactions of a user that are part of the test dataset but were not part of the training dataset (*BBIT4*).

BBIT3: Assert algorithm can predict for a new user. In this test case, we verify if a recommendation algorithm can make recommendations for all users in X^{test} that were not in X^{train}. To that end, we define $X^{\text{toy}} \in \{0,1\}^{|U|\times|I|} \sim \text{Bernoulli}(\rho)$. We then split X^{toy} such that the first K rows make up the training dataset and the final $|U| - K$ rows the test dataset, or formally, $X^{\text{train}} = X^{\text{toy}}_{0:K-1,:}$ and $X^{\text{test}} = X^{\text{toy}}_{K:|U|-1,:}$. We define K such that $K = \phi|U|$ with $\phi < 1$. Now, we fit a model M on X^{train} and request predictions $X^{\text{pred}} = M(X^{\text{test}})$. Then, the bottom $K : |U| - 1$ rows of X^{pred} contain the predictions for the users which were new to the algorithm at prediction time. We assert that

$$\forall u \in K, \ldots, |U| - 1 : \sum_{i\in I} x^{\text{pred}}_{ui} > 0.$$

BBIT4: Assert algorithm can predict using new user interactions. In this test case, we verify if a recommendation algorithm can use new interactions by a user that are part of the test dataset but were not part of the training dataset. With X^{toy} as defined in the previous test, we set the test dataset equal to this toy dataset $X^{\text{test}} = X^{\text{toy}}$ and create the training dataset X^{train} in such a way that $\forall u \in U : X^{\text{train}}_u \subset X^{\text{toy}}_u$. We then train a model and later request recommendations twice, once to obtain $X^{\text{pred}} = M(X^{\text{test}})$ and the second time to obtain $\hat{X}^{\text{pred}} = M(X^{\text{train}})$. We then verify that

$$\forall u \in U : \text{top-K}(X^{\text{pred}}_u) \neq \text{top-K}(\hat{X}^{\text{pred}}_u).$$

3.3.4 BBIT5-6: Can results be reproduced? When running offline experiments with recommendation algorithms, we want to ensure that results are reproducible, i.e., with the exact same setup we can obtain the same results every time. Obtaining reproducible results is not a straightforward task. For example, iterative recommendation algorithms rely on randomization in sampling batches for training or setting initial values. Implementations of recommendation algorithms that aim for reproducibility usually expose a 'seed' parameter that can be used to set the seed of the random numbers generator that is used to initialize parameter values and when sampling batches. Another common source of errors is dropout, i.e., the practice of dropping out some values in the input or hidden layers of a neural architecture to regularize learning and avoid overfitting. While dropout aids in training, applying dropout at prediction time can lead to unintentional stochasticity of predictions. Here, we will propose two tests, one to verify that model prediction is reproducible and the other to verify that model training can be reproduced.

BBIT5: Assert predictions are reproducible. This test is applicable to all deterministic recommendation algorithms, i.e., algorithms where the same model should lead to the same predictions every time. We create a dataset $X^{\text{toy}} \in \{0,1\}^{|U|\times|I|} \sim \text{Bernoulli}(\rho)$, which we will use to both train a model and obtain predictions, $X^{\text{toy}} = X^{\text{train}} = X^{\text{test}}$. We first train the model M on the training dataset and then compute $X^{\text{pred}} = M(X^{\text{test}})$ twice and verify that

$$\forall u \in U : \text{top-K}({X_1}^{\text{pred}}) = \text{top-K}({X_2}^{\text{pred}})$$

BBIT6: Assert training is reproducible. This test is applicable to all recommendation algorithms that expose a 'seed' parameter. The goal of this test is to find out that training with the same training data and identical hyperparameters and seed lead to identical models. However, as this is a black box test, we assume no access to the algorithm's internals. Therefore, to verify that the two recommendation models are identical, we verify that the predictions they make are identical. As such, this test case assumes that the algorithm has passed test case *BBIT5*, which verifies that predictions are reproducible. Formally, let M_1 and M_2 be two models trained on the same training data X^{train}, as defined in the previous test, and using identical hyperparameters and seed parameter. We then make predictions using both models, i.e., we obtain

$$X_1^{\text{pred}} = M_1(X^{\text{train}}) \text{ and } X_2^{\text{pred}} = M_2(X^{\text{train}}).$$

Now, we verify that

$$\forall u \in U : \text{top-K}({X_1}^{\text{pred}}) = \text{top-K}({X_2}^{\text{pred}})$$

3.4 White Box Integration Tests

To verify that the algorithm's design is sound and free of errors, we can use white box integration tests that look at the representations it learned. To this end, we propose two tests. The first, *WBIT1*, verifies that the algorithm is able to learn simple patterns in the training data, in a similar vein to *BBIT1*. The second, verifies that the algorithm does not learn 'self-similarity'. If an algorithm learns that an item is exactly similar to itself, the presence of that item in a user's interaction history will cause the item to be recommended again, which is not always desirable.

3.4.1 WBIT1: Is the algorithm able to learn simple patterns in the training data? To verify whether the recommendation algorithms learn meaningful internal representations, we perform a minimum functionality test in which we give the algorithm a simple pattern in the training dataset to overfit. We propose one test case for each internal representation, i.e., item embeddings and item similarities.

WBIT1Emb: Assert identical items are close in the embedding space. A key assumption in item embedding-based algorithms is that if two items appear in the interaction history of many of the same users, they must be located near each other in the embedding space. To verify this, we create a dataset $A \in \{0,1\}^{|U|\times|I|} \sim \text{Bernoulli}(\rho)$ with ρ the density of the matrix. Next, we construct a second matrix $\hat{A}$ by sampling a subset of N items, i.e., columns, from A, with $N = \psi|I|$ and ψ configurable by the tester. Now, we can define X^{toy} as the concatenation of the two matrices: $X^{\text{toy}} = [A\ \hat{A}]$. We train the model M on X^{toy}, which will learn item embeddings $V \in \mathbb{R}^{|I|\times K}$. Then we verify that the distance between the original item and its duplicate is below a threshold ϵ,

$$\forall i, j \in I^{\text{toy}} \mid X_i^{\text{toy}} = X_j^{\text{toy}} : d(V_i, V_j) < \epsilon.$$

WBIT1Sim: Assert identical items are close neighbors. In item similarity-based algorithms, we find a similar assumption: If two items appear in the interaction history of many of the same users, they must belong to each other's close neighborhoods. Using X^{toy} as defined in the previous test, we assert that

$$\forall i, j \in I^{\text{toy}} \mid X_i^{\text{toy}} = X_j^{\text{toy}} : j \in \text{neighbors}(i) \wedge i \in \text{neighbors}(j).$$

3.4.2 WBIT2: Does the algorithm learn self-similarity? In most use cases, it is not desirable that an algorithm learns that an item is similar to itself. For example, in EASE$^{\text{r}}$ [66], Steck chooses to explicitly require that the diagonal of the similarity matrix consists of all zeros. However, depending on the implementation, an item similarity-based algorithm could learn that an item is similar to itself. For example, in the `scikit-learn` [54] implementation of cosine similarity, the similarity between a vector T and itself, $\text{sim}(T,T) = 1$. Therefore, we propose a test case to verify that in a trained model, no item is similar to itself.

WBIT2Sim: Assert no item is similar to itself. For the last time, we define $X^{\text{toy}} \in \{0,1\}^{|U|\times|I|} \sim \text{Bernoulli}(\rho)$ and train a model M on this dataset. Then, with neighbors as defined in Equation 3, we verify that

$$\forall i \in I : neighbors(i) \cap i = \emptyset.$$

3.5 Black Box Unit Tests

Black box unit tests verify that a function is free of errors based on the signature and functional requirements alone. Black box unit tests are therefore often mentioned in the same context as test-driven development, i.e., when first an appropriate test case is written, after which the same person or a different person implements the actual function. As black box unit tests are very specific to the recommendation algorithm's internals, we will not propose any generic black box unit tests here, but rather give the reader an intuition for how to write such tests themselves. We will use the ItemKNN [16] algorithm as our example. ItemKNN was originally presented with two possible similarity functions: conditional probability and cosine similarity. As such, we can expect that our ItemKNN algorithm implementation will implement a `cosine_similarity(X)` and a `conditional_probability(X)` method to compute similarities between the items, i.e., columns, of X. We will test the correctness of `conditional_probability(X)`. First, we define a minimal X,

$$X = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

Then, we manually compute the expected conditional probability similarity between the items, $\text{sim}(i, j) = \frac{|(\sum_u (x_{ui} \cap x_{uj}))|}{\sum_u x_{ui}}$ and add those to a matrix

$$EP = \begin{bmatrix} 1. & 0. & 1. \\ 0 & 1. & 1. \\ 0.66 & 0.33 & 1. \end{bmatrix}.$$

We then compute $CP = \text{conditional\_probability}(X)$ and verify that $CP = EP$.

3.6 White Box Unit Tests

White box unit tests then assume that we have access to a function's implementation to verify that it is free of errors. Similarly to black box unit test, they are therefore usually defined for each function and algorithm separately. Here, we present two white box unit tests that are applicable to recommendation algorithms that use `PyTorch` [53] to learn a model. The tests verify that gradients are updated at the right time.

3.6.1 WBUT1Iter: Are gradients updated at the right time? Iterative algorithms that use gradient descent compute gradients to learn a better model. However, if the algorithm was ill-specified, parts of the model may stagnate and hamper learning. Therefore, at model fitting time, we want to make sure that all gradients are updated. When we are evaluating the performance of the model at fitting time, typically after every few epochs, we want to make sure that gradients are not updated so that this model evaluation has no influence on the model we learn. Similarly, we do not want the model to compute gradients at prediction time. Therefore we define two test cases that test that either gradients are or are not updated.

WBUT1Iter+: Assert gradients are updated. Typically, an iterative recommendation algorithm learns a model M by refining their internal representation R over the course of N training epochs. In each training epoch, the training dataset is split and fed to the model in batches $B_1, B_2, \ldots$. We will use R_0 to denote the internal representation of the model at the start of model fitting and $R_1, R_2, \ldots$ to denote the internal representations learned after batch 1, 2 and so on. In each of these batches, the model computes the gradients with respect to the batch, $\nabla R_1(B_1)$. After every batch we can therefore verify that all gradients have been updated by asserting that $\nabla R_{i-1}(B_{i-1}) \neq \nabla R_i(B_i)$.

WBUT1Iter-: Assert gradients are not updated. When we are using a model to generate predictions or evaluating the performance of our model after some training epochs have passed, we do not want our model to update its gradients. Here, we use generating predictions as our example. To verify whether gradients are not updated when generating predictions, we run a few batches to obtain $\nabla R_i(B_i)$ and store it in a temporary variable $G := \nabla R_i(B_i)$. We then request recommendations from our model, after which we can verify that $\nabla R_i(B_i) = G$, i.e., $\nabla R_i(B_i)$ was not updated during prediction.

4 RECPACK TESTS

As Kanewala and Bieman [36] pointed out, scientific software, such as recommendation algorithms, is usually not written by software engineers, but by scientists. Not all of these scientists can be expected to be familiar with automated software testing best practices. To support them to adopt the test suite proposed in Section 3, we release `RecPack Tests`, an open-source Python implementation of the test suite. It is licensed under AGPL [20]. The source code can be found on *GitLab*[1].

As discussed in Section 2, a test case consists of test inputs, a test procedure and the expected results, which can be either a fully specified or partial oracle. In `RecPack Tests`, we have two

[1]https://gitlab.com/recpack-maintainers/recpacktests

types of objects: test functions and test fixtures. When the test case verifies the outcome by means of a partial oracle, the test procedure and expected results together make up a 'test function' and take a 'test fixture' as test inputs. When the test case instead verifies a fully specified oracle, the outcome and expected results are tightly linked, and so the test inputs are embedded in the 'test function'. This division of responsibility allows the reuse of test fixtures for multiple test cases. After describing the test functions and test fixtures in Sections 4.1 and 4.2 respectively, we show how to use them to create test cases for your recommendation algorithms in Section 4.3.

4.1 Test Functions

As mentioned in Section 3, we assume all algorithms implement a `fit` and `predict` method. This interface was popularized by `scikit-learn` [54] and is used in `RecPack` [47]. Most implementations of recommendation algorithms have similar interfaces, e.g., `Surprise` [32] and `PyLensKit` [17], which can be easily transformed to the `scikit-learn` interface using a wrapper. Further, we assume that both methods take a `RecPack InteractionMatrix` [47] as input. An example of a wrapper can be found in Appendix A.

In our white box tests, we need to make a few additional assumptions. We assume all item embeddings are stored in a variable `item_embeddings_` as a `NumPy ndarray` and item similarities are stored in a variable `item_similarities_` as a `SciPy Sparse CSR Matrix`. Most algorithm implementations in `RecPack` already conform to this interface. Making other packages conform should not require much more than creating a new property with the correct name and possibly a transformation from one storage format to another, e.g., from `Torch Tensor` to `Numpy ndrray`.

Test functions for system and integration level tests use one of three interfaces:

- **System tests** use the interface `assert_ (model, train_data, Optional[])`. They take as inputs a trained `model` and the data it was trained on, `train_data`, and optionally some configuration parameters for the test, e.g., the seed.
- **Integration tests** that rely on a partial oracle use the interface `assert_ (algorithm, toy_data, Optional[])`. They take as inputs an `algorithm` instance and a toy dataset, `toy_data`, as well as optionally some configuration parameters.
- **Integration tests** that rely on a fully specified oracle use the interface `assert_ (algorithm, Optional[])`. Here, the toy dataset is embedded in the test function.

4.2 Test Fixtures

In Section 3, we defined toy datasets for all integration tests. We provide implementations of each of these toy datasets in `RecPack Tests`. Each of them comes with default values for the configurable parameters, which can be overwritten by the tester. For system tests, the dataset, i.e., the test fixture, needs to be supplied the tester.

BBIT3, BBIT4, BBIT5, BBIT6, WBIT1. This test fixture is used by integration tests that do not require specific patterns to be present in the dataset. We sample $X^{\text{toy}} \in \{0,1\}^{|U|\times|I|} \sim \text{Bernoulli}(\rho)$, with $|U| = 100$, $|I| = 20$ and $\rho = 0.35$.

BBIT1TopK. This test fixture constructs X^{toy} as described in Section 3.3.1, i.e.,

$$X^{\text{toy}} = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}.$$

We sample $A, B \sim \text{Bernoulli}(\rho)$ with $\frac{|U|}{2} = 300$, $\frac{|I|}{2} = 4$ and $\rho = 0.4$ such that X^{toy} is of size 600×8.

BBIT1Seq. This test fixture creates $\mathcal{S}^{\text{toy}}$ as described in 3.3.1. In this test case we assign to every user a sub-sequence of one of two proto-sequences, S^A and S^B. We set the length of the proto-sequences $n = 5$ and the number of users assigned a sub-sequence of each of the proto-sequences $|U^A| = |U^B| = 300$. The final toy dataset $\mathcal{S}^{\text{toy}}$ contains 600 sequences.

BBIT2TopK-A. This test fixture constructs X^{toy} as described in Section 3.3.2, i.e.,

$$X^{\text{toy}} = \begin{bmatrix} A \\ \tilde{A} \end{bmatrix}.$$

where $A \sim \text{Bernoulli}(\rho)$ with $|U| = 100$ and $\tilde{A}$ is constructed by sampling a subset of items for each copy of a user with $\psi = 0.9$, such that 10% of a user's interaction history is removed.

BBIT2TopK-B. This test fixture uses the fixture created for *BBIT2TopK-A*, but then adds $I_u^{\text{rand}} \sim \text{Mult}(n_u, |I|, P_{|I|})$ to the interaction history of every copy of a user with $n_u = 0.1|I_u|$ such that every user and their copy have the same number of interactions in their interaction history.

BBIT2Seq-A. This test fixture constructs $\mathcal{S}^{\text{toy}}$ as described in 3.3.2. First, we sample $|U|$ sequences $S \sim \text{Mult}(n, |I|, P_{|I|})$, also with $|U| = 100$ and with $1 \leq n_u \leq 20$. Once again we set $\psi = 0.9$ such that 10% of a user's interaction history is removed.

BBIT2Seq-B. This test fixture uses the fixture created for *BBIT2Seq-A*, but then adds $S_u^{\text{rand}} \sim \text{Mult}(n_u - k, |I|, P_{|I|})$ to the interaction history of every copy of a user with $k = \psi n_u$ such that every user and their copy have the same number of interactions in their interaction history.

4.3 Using RecPack Tests

`RecPack Tests` provides test functions and fixtures, but leaves defining the relevant test cases for their algorithms to the tester. In this Section we illustrate how to define test cases with `RecPack Tests` and run them using `pytest` [38]. In our example we test the ItemKNN algorithm, both with the conditional probability and cosine similarity function, using four system-level black box tests.

4.3.1 Step 1: Defining pytest fixtures. First, we define both algorithms and datasets as a `pytest fixture` such that we can reuse them in multiple test cases. We create a fixture for a random dataset using the `RecPack DummyDaset` and one for the MovieLens Latest Small dataset. Then, we create a parameterized fixture that returns an instance of the ItemKNN algorithm. A parameterized will generate a test case for each of the parameters specified.

```
import pytest
from recpack.algorithms import ItemKNN
from recpack.datasets import DummyDataset, MovieLensLatestSmall

# We use scope=session such that the dataset is created once
# for the entire test suite
@pytest.fixture(scope="session")
def dummy_dataset():
    return DummyDataset(seed=12345).load()


@pytest.fixture(scope="session")
def ml_latest_small():
    return MovieLensLatestSmall().load()


# We parametrize the fixture,
# such that we cover both conditional probability
```

```
# and cosine similarity to achieve full branch coverage
@pytest.fixture(
    scope="session",
    params=[
        {"K": 20, "similarity": "cosine"},
        {"K": 20, "similarity": "conditional_probability"}
    ]
)
def item_knn(request):
    return ItemKNN(**request.param)

# Next we create two trained model fixtures.
# Each will evaluate to two separate instances
# one for each parametrization of ItemKNN
@pytest.fixture(scope="session")
def item_knn_ml_model(item_knn, ml_latest_small):
    item_knn.fit(ml_latest_small)
    return item_knn, ml_latest_small

@pytest.fixture(scope="session")
def item_knn_dd_model(item_knn, dummy_dataset):
    item_knn.fit(dummy_dataset)
    return item_knn, dummy_dataset
```

4.3.2 Step 2: Defining the test cases. Given these fixtures, we can now define the test cases using the test functions provided by Recpack `Tests`. We define 4 test cases, corresponding to *BBST1+*, *BBST1-*, *BBST2* and *BBST3*.

```
from recpacktests.system_level.black_box import (
    assert_bbst1plus, assert_bbst1min, assert_bbst2, assert_bbst3)


def test_bbst1plus_ml(item_knn_ml_model):
    assert_bbst1plus(
        item_knn_ml_model[0], item_knn_ml_model[1]
    )


def test_bbst1min_ml(item_knn_ml_model):
    assert_bbst1min(
        item_knn_ml_model[0], item_knn_ml_model[1]
    )

def test_bbst2_ml(item_knn_ml_model):
    assert_bbst2(
        item_knn_ml_model[0], item_knn_ml_model[1]
    )


def test_bbst3_ml(item_knn_ml_model):
    assert_bbst3(
        item_knn_ml_model[0], item_knn_ml_model[1]
    )

def test_bbst1plus_dd(item_knn_dd_model):
    assert_bbst1plus(
        item_knn_dd_model[0], item_knn_dd_model[1]
```

```
    )


def test_bbst1min_dd(item_knn_dd_model):
    assert_bbst1min(
        item_knn_dd_model[0], item_knn_dd_model[1]
    )

def test_bbst2_dd(item_knn_dd_model):
    assert_bbst2(
        item_knn_dd_model[0], item_knn_dd_model[1]
    )


def test_bbst3_dd(item_knn_dd_model):
    assert_bbst3(
        item_knn_dd_model[0], item_knn_dd_model[1]
    )
```

4.3.3 Step 3: Running and analysing the results. Finally, we run the test cases using

```
pytest test_packages/test_example.py
```

and obtain the following result

```
====================== 16 passed, 10 warnings in 2.78s ======================
```

which indicates all tests passed.

If instead of *BBST2* we had run *BBST2*^{c}, we would have obtained the following output

```
============================ short test summary info ============================
FAILED test_packages/test_example.py::test_bbst2c_ml[item_knn0] -
AssertionError: Algorithm recommended an item a user previously interacted with.
FAILED test_packages/test_example.py::test_bbst2c_dd[item_knn0] -
AssertionError: Algorithm recommended an item a user previously interacted with.
FAILED test_packages/test_example.py::test_bbst2c_ml[item_knn1] -
AssertionError: Algorithm recommended an item a user previously interacted with.
FAILED test_packages/test_example.py::test_bbst2c_dd[item_knn1] -
AssertionError: Algorithm recommended an item a user previously interacted with.
================== 4 failed, 12 passed, 10 warnings in 3.34s ==================
```

indicating which test case failed for which dataset and which algorithm.

5 EVALUATING RECPACK TESTS

In this Section, we evaluate the usefulness of our proposed test suite, and the accompanying open-source implementation, `RecPack Tests`. To do so, we use `RecPack Tests` to test the correctness of recommendation algorithms implemented in four popular Python packages for recommender systems, `RecPack` [47], `PyLensKit` [17], `Surprise` [32] and `Cornac` [63]. We chose these packages because their interfaces are very similar to the interface of `RecPack`, i.e., the `scikit-learn` interface. As the name suggests, `RecPack Tests` was created by the authors of `RecPack`.

The three packages are also among the most starred recommender systems Python packages on GitHub[2,3,4]. We test the latest release, at the time of writing, of each of the packages. The test results for each of the packages are reported in Sections 5.1, 5.2, 5.3 and 5.4 respectively.

For each of the four packages, we evaluate three aspects of the proposed test suite. Firstly, we require that we have at least one test case for every implicit feedback collaborative filtering recommendation algorithm in the package. This gives an indication of how broadly applicable our proposed test suite is. Secondly, we measure the completeness of the proposed test suite by means of its statement coverage. In Section 2.1, we saw that statement coverage is the weakest coverage we may require of a program. Nonetheless, in the absence of viable alternatives, it is a useful requirement. Thirdly, we make note of whether we are able to uncover bugs, undocumented functional requirements or design flaws. Finally, we report some overarching conclusions in Section 5.5.

To run the system level tests, we require a real dataset. For this, we make use of the 'MovieLens Latest Small Dataset' [28], as provided with `PyLensKit` under the MIT License. This dataset contains 100, 004 ratings by 671 users on 9, 125 movies between January 09, 1995 and October 16, 2016.

5.1 RecPack

`RecPack` [47] is a Python package that focuses on Top-N recommendation with implicit feedback data. We test version 0.3.5 of `RecPack`. An overview of the recommendation algorithms implemented in the package and their properties can be found in Table 3. We find that our black box system tests are applicable to all of `RecPack`'s algorithms and all personalized recommendation algorithms have at least one property that can be tested using our test suite. Although GRU4Rec and BPRMF both learn an item embedding, they are currently stored in a way that is different from all other item embedding-based algorithms in `RecPack`. As a result, we do not run *WBITEmb* on these algorithms. For algorithms that have clear 'branches', i.e. they execute disjoint code paths depending on the value of a hyperparameter, we run the test suite for every value of the hyperparameter. For iterative algorithms that rely on stochastic gradient descent to learn a model we run an initial grid search to determine a reasonable value for the batch size, number of training epochs and learning rate and use those for all algorithms and all test cases. For hyperparameters that are unique to an algorithm we manually determine a reasonable value.

In Table 4 we report test results aggregated per algorithm for `RecPack`. We refer the interested reader to Appendix B for the full test results. We run a total of 375 test cases, resulting in 43 failures and 332 successes. We achieve a total statement coverage of 85%.

The test suite allows us to uncover the following bugs[5]:

- Multiple failures on the BBIT6 test case indicate that training is not reproducible from some of the `RecPack` algorithm implementations. Algorithms that use `TorchMLAlgorithm` (GRU4Rec, RecVAE, MultVAE and Prod2Vec) as their base class set the global seed for the `PyTorch` random numbers generator, which causes issues when multiple algorithms attempt to set this same global seed.
- Further, attempting to run *BBIT6* allows us to uncover that two iterative algorithms do not expose a seed parameter (SLIM and WMF) and so experiments with these algorithms are not reproducible.

[2]https://github.com/NicolasHug/Surprise,
[3]https://github.com/lenskit/lkpy
[4]https://github.com/PreferredAI/cornac
[5]All of these bugs are fixed in `RecPack` version 0.3.6.

Algorithm	Task	Item Representation	Learning Method
BPRMF [57]	Top-K	Embeddings	Iterative
$EASE^R$ [66]	Top-K	Similarities	-
GRU4Rec [30]	Sequential	-	Iterative
ItemKNN [16]	Top-K	Similarities	-
MultVAE [42]	Top-K	-	Iterative
NMF [14]	Top-K	Embeddings	-
NMFItemToItem [14]	Top-K	Similarities	-
Popularity	-	-	-
Prod2Vec [25]	Top-K	Similarities	Iterative
Random	-	-	-
RecVAE [65]	Top-K	-	Iterative
SLIM [52]	Top-K	Similarities	-
TARSItemKNN [43]	-	Similarities	-
Sequential Rules [44]	Sequential	Similarities	-
SVD [27]	Top-K	Embeddings	-
SVDItemToItem [27]	Top-K	Similarities	-
STAN [22]	Sequential	-	-
WMF [31]	Top-K	Embeddings	-

Table 3. Overview of the properties of algorithms included in RecPack that determine which tests can be run. A '-' indicates the algorithm implementation has a task, item representation or learning method for which there is no test in the test suite.

- BPRMF fails *BBIT4*, but not *BBIT3*. This allows us to uncover that BPRMF does not set the user embeddings of users who were not seen in training to zero. Because of this, the algorithm makes random recommendations for these users.

Not all test failures can be attributed to bugs in the implementation. These are:

- STAN and GRU4Rec fail *BBIT1Seq*, which verifies whether the algorithm can learn simple sequential patterns in the data.
- TARSItemKNN fails *BBST1+* and *WBST1Sim+* which indicates a failure to recommend for all users we expect the algorithm to. This is due to the fact that the algorithm has not learned neighbors for all items. We speculate that this can be attributed to the discounting of older interactions in the model.
- Both SVD and NMF fail test cases *BBIT3* and *BBIT4*, indicating that they cannot use new interactions at prediction time. Both algorithms compute user embeddings that are not updated at prediction time. The inability of both algorithms to recommend for new users or using new interactions is currently not documented and so the documentation should be updated to reflect this limitation.
- BPRMF is the only algorithm to fail *BBST3*, which indicates that the seed has a significant impact on the performance of the algorithm.
- NMF and RecVAE fail *BBIT2TopK-A* and *BBIT2TopK-B* whereas Prod2VecClustered fails *BBIT2TopK-A*. Similarly, GRU4Rec fails both *BBIT2Seq-A* and *BBIT2Seq-B*. This indicates that these algorithms do not strongly personalize recommendations and similar users may receive exactly the same recommendations.

Algorithm	#Successes	#Tests Run	Success Rate
BPRMF	9	13	69.23
EASE	14	14	100.00
GRU4RecCrossEntropy	9	13	69.23
GRU4RecNegSampling	9	13	69.23
ItemKNN	56	56	100.00
MultVAE	12	13	92.31
NMF	10	14	71.43
NMFItemToItem	15	15	100.00
Popularity	6	7	85.71
Prod2Vec	45	51	88.24
Prod2VecClustered	13	16	81.25
Random	6	6	100.00
RecVAE	10	13	76.92
SLIM	39	42	92.86
STAN	9	10	90.00
SVD	11	14	78.57
SVDItemToItem	15	15	100.00
SequentialRules	13	14	92.86
TARSItemKNN	23	28	82.14
WMF	9	10	90.00

Table 4. Summary of test results for all RecPack algorithms. For each algorithm we report the number of successful tests, the total number of tests run and the success rate.

5.2 PyLensKit

`PyLensKit` [17] is the successor to the Java version of `LensKit`. It contains both algorithms for rating prediction and Top-N recommendation with implicit feedback data. We test version 0.14.2 of `PyLensKit`. Our experiments focus on the algorithms that work with implicit feedback data. We run both the black and white box tests included in the test suite, because `PyLensKit` allows easy access to internal item representations. As none of the algorithms use `PyTorch`, we cannot run the white box unit tests. Additionally, we cannot run *BBIT3* and *BBIT4*, as `PyLensKit` memorizes the training dataset and at prediction time represents a user solely by their user ID. In Table 5 we list all of `PyLensKit` algorithms that are tested.

Algorithm	Task	Item Representation	Learning Method
ImplicitMF [31]	Top-K	Embeddings	-
ItemKNN [16]	Top-K	Similarities	-
Popular	-	-	-
UserKNN [8]	Top-K	-	-

Table 5. Overview of the properties of algorithms included in `PyLensKit` that determine which tests can be run. A '-' indicates the algorithm implementation has a task, item representation or learning method for which there is no test in the test suite.

For PyLensKit a total of 37 test cases are run, of which 3 fail and 34 succeed. The summary of these results is presented in Table 6, the interested reader can find the full results in Appendix B. We reach a coverage of the lenskit.algorithms module of 40%. Note that PyLensKit contains both algorithms that use implicit feedback data and rating data, and the latter are not tested.

Algorithm	#Successes	#Tests Run	Success Rate
ImplicitMF	10	12	83.33
ItemKNN	11	12	91.67
Popular	5	5	100.00
UserKNN	8	8	100.00

Table 6. Summary of test results for all PyLensKit algorithms. For each algorithm we report the number of successful tests, the total number of tests run and the success rate.

We find the following interesting failures:

- ImplicitMF fails *BBIT6*, which indicates that setting the seed does not guarantee reproducible training.
- ItemKNN fails *WBST1Sim-* and ImplicitMF fails *WBST1Emb-*, indicating that items that were not in the training dataset have neighbors and nonzero embeddings. However, because PyLensKit memorizes users, this can never affect results negatively.

5.3 Surprise

Where the previously discussed packages support or focus on Top-K recommendation with implicit feedback data, Surprise [32] only supports rating prediction. In order to use the package with implicit feedback data we had to transform the implicit feedback data to rating data. We decide to provide binary ratings, i.e., a rating of either 0 or 1. All items a user interacted with are assumed to be rated 1 by the user. All items a user did not interact with are assumed to be rated 0. We test version 1.1.3 of Surprise. The algorithms we test and their properties can be found in Table 7. For both ItemKNN and UserKNN, we generate test cases for all supported similarity functions and as such cover all code branches. Because the algorithm implementations in Surprise take a very long time to fit a model and make predictions, we use the DummyDataset included with RecPack for system level tests, rather than the MovieLens Latest Small Dataset.

Algorithm	Task	Item Representation	Learning Method
CoClustering [23]	Top-K	-	-
ItemKNN [16]	Top-K	Similarities	-
NMF [14]	Top-K	Embeddings	-
SVD [27]	Top-K	Embeddings	-
SVDpp [27]	Top-K	Embeddings	-
UserKNN [8]	Top-K	-	-

Table 7. Overview of the properties of algorithms included in Surprise that determine which tests can be run. A '-' indicates the algorithm implementation has a task, item representation or learning method for which there is no test in the test suite.

Algorithm	#Successes	#Tests Run	Success Rate
CoClustering	5	9	55.56
ItemKNN	38	48	79.17
UserKNN	28	32	87.50
NMF	10	12	83.33
SVD	7	12	58.33
SVDpp	7	12	58.33

Table 8. Summary of test results for all `Surprise` algorithms. For each algorithm we report the number of successful tests, the total number of tests run and the success rate.

We execute a total of 125 test cases of which 30 fail and 95 succeed. Results are summarized by algorithm in Table 8. Full test results can be found in Appendix B The tests reach a statement coverage of 54% of the `surprise.prediction_algorithms` module. We find that most of the uncovered statements pertain to code paths that perform transformations of rating data, e.g., centering ratings on a scale of 1-5.

Our test run results in the following failures, most of which are due to the transformation of implicit feedback data into rating data:

- Both SVD and SVDpp fail *BBIT1TopK*, which verifies if the algorithm can learn a simple pattern in the data, and *WBIT1Emb*, which verifies if identical items have similar item embeddings. Although these failure can be due to our transformation of the implicit feedback data into rating data, we would expect the algorithm to pass *WBIT1Emb*.
- Similarly, CoClustering and NMF fail *BBIT2TopK-A* and *BBIT2TopK-B*. In the case of CoClustering, this can be due to the amount of clusters: If similar users are in the same cluster, they get the same recommendations.
- ItemKNN fails *WBIT2Sim* because it does not remove self-similarity and as a result, each item is its own closest neighbor.
- ItemKNN also fails *WBST1Sim-*, and both SVD and SVDpp fail *WBST1Emb-* due to our transformation of the implicit feedback data into rating data, which created explicit 0 ratings.
- For cosine similarity, the similarity computation will throw a DivisionByZero error when one of the users has only 0 ratings in the dataset. This causes *WBST1Sim+* and *BBST1-* to fail.
- SVD, SVDpp and CoClustering fail *BBST3* indicating that results are heavily dependent on the random seed. However, as we used the DummyDataset, which contains random interactions instead of a real dataset, additional variation is to be expected as there are no patterns in the data which the algorithm can learn.

5.4 Cornac

`Cornac` [63] is a comparative framework for multi-modal recommendation algorithms. Itt contains a number of collaborative filtering recommendation algorithms that can learn a model using implicit feedback data only. We test version 1.14.2 of `Cornac`. The list of algorithms we test is presented in Table 9. We cannot not run the proposed white box unit tests because `Cornac` does not use `PyTorch`. White box system and integration tests are also not run because internal item representations are stored differently between the different algorithms, which makes it difficult to automate testing. As we did for the other packages, if an algorithm executes a different code path based on the value of a hyperparameter, we make sure to cover each code path and report results across hyperparameter values. For example, VAECF has a hyperparameter to determine the

Algorithm	Task	Item Representation	Learning Method
BPR [57]	Top-K	Embeddings	Iterative
BiVAECF [69]	Top-K	-	Iterative
GMF [29]	Top-K	Embeddings	Iterative
IBPR [41]	Top-K	Embeddings	Iterative
ItemKNN [16]	Top-K	Similarities	-
MF [37]	Top-K	Embeddings	Iterative
MLP [29]	Top-K	Embeddings	Iterative
MMMF [75]	Top-K	Embeddings	Iterative
MostPop	-	-	-
NMF [14]	Top-K	Embeddings	Iterative
NeuMF [29]	Top-K	Embeddings	Iterative
PMF [48]	Top-K	Embeddings	Iterative
SKMeans [62]	Top-K	-	-
UserKNN [8]	Top-K	-	-
VAECF [42]	Top-K	-	Iterative
WBPR [21]	Top-K	Embeddings	Iterative
WMF [31]	Top-K	Embeddings	Iterative

Table 9. Overview of the properties of algorithms included in `Cornac` that determine which tests can be run. A '-' indicates the algorithm implementation has a task, item representation or learning method for which there is no test in the test suite.

activation function (sigmoid, tanh, elu, relu and relu6) and one to decide the likelihood (Poisson, Bernoulli, Gaussian). Therefore we run eight tests in total to obtain the highest possible statement coverage.

We run a total of 498 tests of which 396 succeed and 102 fail. The aggregated results for these tests are presented in Table 10. The exact statement coverage for `Cornac` is difficult to measure as many algorithms are implemented using Cython and wrappers around C++ code. The statement coverage of the Python code in the `models` module is 30%. Note that we are unable to test algorithms that require side-info and the various available options for handling side-info.

We identify the following failures:

- BPR fails *BBIT2TopK-A* and *BBIT2TopK-B*, which indicates that results are not strongly personalized and similar users receive exactly the same recommendations. BiVAECF also fails *BBIT2TopK-A* and *BBIT2TopK-B*, but only for some activation functions and the Gaussian likelihood. Other hyperparameter values do succeed these tests. Similarly, VAECF fails for the Poisson, Bernoulli and Gaussian likelihood and sigmoid and relu6 activation functions.
- MLP, NeuMF and GMF fail *BBIT6*, indicating that training is not reproducible.
- BiVAECF, VAECF, MLP, NeuMF, PMF, IBPR, WMF, NMF, MMMF, MF, WBPR and GMF all fail *BBIT1TopK*, i.e., they are unable to learn the simple pattern in the data. This could be due to poor choices of hyperparameters, although the same hyperparameters did work when used for implementations of the same algorithm in `RecPack`.
- For MLP, NeuMF and GMF, the Adagrad and SGD optimizers fail *BBST3*, whereas the Adam optimizer does not. This suggests that the Adam optimizers is less prone to getting stuck in sub-optimal local minima for these algorithms.

Algorithm	#Successes	#Tests Run	Success Rate
BPR	6	9	66.67
BiVAECF	59	72	81.94
GMF	26	36	72.22
IBPR	6	8	75.00
ItemKNN	8	8	100.00
MF	28	36	77.78
MLP	61	81	75.31
MMMF	8	9	88.89
MostPop	5	6	83.33
NMF	14	18	77.78
NeuMF	68	90	75.56
PMF	16	18	88.89
SKMeans	9	9	100.00
UserKNN	8	8	100.00
VAECF	58	72	80.56
WBPR	8	9	88.89
WMF	8	9	88.89

Table 10. Summary of test results for all Cornac algorithms. For each algorithm we report the number of successful tests, the total number of tests run and the success rate.

5.5 Discussion

5.5.1 Shared Algorithms. Some algorithms are implemented in more than one package, e.g., ItemKNN is implemented in all four packages and BPRMF in both RecPack and Cornac. Interestingly, two implementations of the same algorithms sometimes expose different hyperparameters. When a hyperparameter is shared between different implementations of an algorithm, we made sure to set them to the same values to allow comparison. We find that different implementations do not always succeed at the same tests.

For example, MultVAE/VAECF is implemented in both RecPack and Cornac. Whereas the RecPack implementation succeeded 12 out of 13 tests, the Cornac implementation succeeded only 58 out of 72 tests. The much larger number of tests run on Cornac is due to the large amount of hyperparameter values that can be chosen for the activation function between hidden layers and likelihood, whereas RecPack sets the activation function to 'tanh' and the likelihood to Multinomial, as in the original paper [42]. Evidently, different choices of activation and likelihood function result in sub-par performance of the algorithm. On the other hand, BPRMF, NMF and WMF are also implemented in both RecPack and Cornac and succeed a similar percentage of tests.

These findings highlight that automated testing of the correctness of recommendation algorithms can contribute interesting insights on the performance of recommendation algorithms.

5.5.2 Test Case Success Rates. In Table 11, we present an overview of how often a test case was run and how successful they were. In general, we notice that our black box tests could be applied to many more algorithms than our white box tests. This is of course to be expected, as white box tests rely on the algorithm's internal item representations and as such could not be run for all packages. Because only RecPack contains some sequential recommendation algorithms, the test cases developed specifically for this recommendation task, *BBIT2Seq-A*, *BBIT2Seq-B*, *BBIT1Seq* were used only a few times. However, it is interesting to note that half of the times these test

Test ID	#Successes	#Runs	Success Rate
BBST1+	98	100	98.000000
BBST1-	93	100	93.000000
BBST2	96	96	100.000000
BBST2^c	4	4	100.000000
BBST3	49	70	70.000000
WBST1Emb+	3	6	50.000000
WBST1Emb-	6	6	100.000000
WBST1Sim+	19	22	86.363636
WBST1Sim-	17	22	77.272727
BBIT1TopK	44	90	48.888889
BBIT1Seq	1	4	25.000000
BBIT2TopK-A	77	92	83.695652
BBIT2TopK-B	79	92	85.869565
BBIT2Seq-A	2	4	50.000000
BBIT2Seq-B	2	4	50.000000
BBIT3	26	28	92.857143
BBIT4	24	28	85.714286
BBIT5	99	99	100.000000
BBIT6	60	99	60.606061
WBIT1Emb	4	6	66.666667
WBIT1Sim	17	22	77.272727
WBIT2Sim	18	22	81.818182
WBUT1Iter+	9	9	100.000000
WBUT1Iter-	9	9	100.000000

Table 11. Summary of test results aggregated by test case. For each test case we report the number of successful tests, the total number of times it was run and the success rate.

cases were run, they failed, indicating potential design flaws in the sequential recommendation algorithms tested. We also notice a lot of failures of *BBIT1TopK*. Most of these can be attributed to `Cornac`, where many matrix factorization algorithms failed this test for multiple hyperparameter values. The cause of these failures is not immediately obvious. Interestingly, simple algorithms such as ItemKNN and EASE^r do consistently succeed at this test case. Many algorithms are also sensitive to the random seed used, as evidenced by the success rate of *BBST3*. Finally, we note that many algorithms fail *BBIT6*. Clearly, implementing an algorithm in such a way that results can be reproduced exactly is a difficult task. Overall, we have demonstrated that our test suite can be applied broadly and is able to uncover both bugs, as well as interesting behaviors that could possibly be attributed to design flaws.

6 CONCLUSION AND FUTURE WORK

In this work we propose an automated test suite to evaluate the correctness of implicit feedback, collaborative filtering recommendation algorithms. Using the test suite, researchers can identify potential issues with the algorithms that can affect their performance. Test cases were designed drawing from our experience building `RecPack` and using techniques encountered in the prior work on testing machine learning systems. We stress that automated testing can provide a complementary perspective to traditional methods for evaluation. To facilitate the adoption of testing in the recommender systems community, we release `RecPack Tests`, an open-source implementation of the test suite proposed. We show the efficacy of `RecPack Tests` by testing the correctness of algorithm implementations in four popular Python packages for recommender systems, `RecPack`, `PyLensKit`, `Surprise` and `Cornac`. In doing so, we demonstrate the importance of testing recommendation algorithms: Our test suite was able to identify and uncover interesting (undesired) behaviors and even bugs in these popular open-source packages. We also show that `Recpack Tests` can be broadly used, as we are able to test at least one property of each of the algorithms tested, across all four open-source packages.

This work is only a first step towards comprehensive testing of recommendation algorithms. We identify several interesting directions for future research based in our discussion of testing machine learning systems in Section 2.4. A first has to do with extending the test suite in several directions. Firstly, the test suite can be extended with more (complex) test cases for the same task, item representation and learning method to obtain a greater measure of 'test completeness'. Next, differential testing could provide additional insight into how implementations of the same algorithm differ and could lead to the discovery of incorrect behavior that cannot be captured with the property-based and metamorphic tests proposed. Then, the test suite can be further extended to cover more (specific) use cases. For example, the requirements for recommendations in e-commerce may be different from those in the music domain and specific test cases can be developed for each. The test suite can also be extended to other recommendation tasks than top-K and sequential recommendation and to other internal representations than item embeddings and item similarities. Many more representations exists that are not covered here, e.g., user embeddings and the bottleneck layers typically encountered in auto-encoder architectures.

Another direction could be to invest in the development of formal completeness guarantees for test suites. The test cases proposed here were developed using error guessing and by identifying promising techniques in earlier works on testing machine learning systems. Our test suite therefore comes with no guarantees of completeness.

We can also look at applying testing to other properties of recommendation algorithms beyond correctness. Test suites could be defined to automatically examine the robustness of algorithms under attack, their explainability, fairness, or any of the properties identified by Zhang et al. [77].

Finally, we can define similar test suites for other types of recommendation algorithms than purely collaborative filtering algorithms, such as content-based, knowledge based, or hybrid techniques.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their insightful feedback and their valuable suggestions for future work.

REFERENCES

[1] Technical Committee ISO/IEC JTC 1. 2017. ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary. , 541 pages. https://doi.org/10.1109/IEEESTD.2017.8016712

[2] Vito Walter Anelli, Alejandro Bellogin, Antonio Ferrara, Daniele Malitesta, Felice Antonio Merra, Claudio Pomo, Francesco Maria Donini, and Tommaso Di Noia. 2021. Elliot: A Comprehensive and Rigorous Framework for Reproducible Recommender Systems Evaluation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Virtual Event Canada, 2405–2414. https://doi.org/10.1145/3404835.3463245

[3] Vito Walter Anelli, Alejandro Bellogín, Tommaso Di Noia, Dietmar Jannach, and Claudio Pomo. 2022. Top-N Recommendation Algorithms: A Quest for the State-of-the-Art. In *Proceedings of the 30th ACM Conference on User Modeling, Adaptation and Personalization*. ACM, Barcelona Spain, 121–131. https://doi.org/10.1145/3503252.3531292

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] Joeran Beel, Marcel Genzmehr, Stefan Langer, Andreas Nürnberger, and Bela Gipp. 2013. A Comparative Analysis of Offline and Online Evaluations and Discussion of Research Paper Recommender System Evaluation. In *Proceedings of the International Workshop on Reproducibility and Replication in Recommender Systems Evaluation - RepSys '13*. ACM Press, Hong Kong, China, 7–14. https://doi.org/10.1145/2532508.2532511

[6] Joeran Beel, Stefan Langer, Marcel Genzmehr, Bela Gipp, Corinna Breitinger, and Andreas Nürnberger. 2013. Research Paper Recommender System Evaluation: a Quantitative Literature Survey. In *Proceedings of the International Workshop on Reproducibility and Replication in Recommender Systems Evaluation - RepSys '13*. ACM Press, Hong Kong, China, 15–22. https://doi.org/10.1145/2532508.2532512

[7] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. 2017. The ML test score: A rubric for ML production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, Boston, MA, 1123–1132. https://doi.org/10.1109/BigData.2017.8258038

[8] John S. Breese, David Heckerman, and Carl Kadie. 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence* (Madison, Wisconsin) *(UAI'98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 43–52.

[9] Pablo Castells and Alistair Moffat. 2022. Offline recommender system evaluation: Challenges and new directions. *AI Magazine* 43, 2 (June 2022), 225–238. https://doi.org/10.1002/aaai.12051

[10] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. 2018. Adversarial Attacks and Defences: A Survey. https://doi.org/10.48550/arXiv.1810.00069

[11] T Y Chen. 1998. *Metamorphic Testing: New Approach for Generating Next Test Cases*. Technical Report. Department of Computer Science, The Hong Kong University of Science and Technology.

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2019. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1 (Jan. 2019), 1–27. https://doi.org/10.1145/3143561

[13] Patrick John Chia, Jacopo Tagliabue, Federico Bianchi, Chloe He, and Brian Ko. 2022. Beyond NDCG: Behavioral Testing of Recommender Systems with RecList. In *Companion Proceedings of the Web Conference 2022*. ACM, Virtual Event, Lyon France, 99–104. https://doi.org/10.1145/3487553.3524215

[14] Andrzej Cihocki and Anh-Huy Phan. 2009. Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E92.A, 3 (2009), 708–721. https://doi.org/10.1587/transfun.E92.A.708

[15] European Commission. 2019. Ethics Guidelines for Trustworthy AI. https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai.

[16] Mukund Deshpande and George Karypis. 2004. Item-Based Top-N Recommendation Algorithms. *ACM Trans. Inf. Syst.* 22, 1 (jan 2004), 143–177. https://doi.org/10.1145/963770.963776

[17] Michael D. Ekstrand. 2020. LensKit for Python: Next-Generation Software for Recommender Systems Experiments. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. ACM, Virtual Event Ireland, 2999–3006. https://doi.org/10.1145/3340531.3412778

[18] Maurizio Ferrari Dacrema, Simone Boglio, Paolo Cremonesi, and Dietmar Jannach. 2021. A Troubling Analysis of Reproducibility and Progress in Recommender Systems Research. *ACM Transactions on Information Systems* 39, 2 (April 2021), 1–49. https://doi.org/10.1145/3434185

[19] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. 2019. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems*. ACM, Copenhagen Denmark, 101–109. https://doi.org/10.1145/3298689.3347058

[20] Free Software Foundation. 2016. GNU Affero General Public License Version 3 (AGPL-3.0). Accessed 26 July 2022.

[21] Zeno Gantner, Lucas Drumond, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2011. Personalized Ranking for Non-Uniformly Sampled Items. In *Proceedings of the 2011 International Conference on KDD Cup 2011 - Volume 18 (KDDCUP'11)*. JMLR.org, 231–247.

[22] Diksha Garg, Priyanka Gupta, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. 2019. Sequence and Time Aware Neighborhood for Session-Based Recommendations: STAN. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Paris, France) *(SIGIR'19)*. Association for Computing Machinery, New York, NY, USA, 1069–1072. https://doi.org/10.1145/3331184.3331322

[23] Thomas George and Srujana Merugu. 2005. A Scalable Collaborative Filtering Framework Based on Co-Clustering. In *Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM '05)*. IEEE Computer Society, USA, 625–628. https://doi.org/10.1109/ICDM.2005.14

[24] Don Gotterbarn, Keith Miller, and Simon Rogerson. 1997. Software Engineering Code of Ethics. *Commun. ACM* 40, 11 (nov 1997), 110–118. https://doi.org/10.1145/265684.265699

[25] Mihajlo Grbovic, Vladan Radosavljevic, Nemanja Djuric, Narayan Bhamidipati, Jaikit Savla, Varun Bhagwan, and Doug Sharp. 2015. E-Commerce in Your Inbox: Product Recommendations at Scale. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 1809–1818. https://doi.org/10.1145/2783258.2788627

[26] Ihsan Gunes, Cihan Kaleli, Alper Bilge, and Huseyin Polat. 2014. Shilling attacks against recommender systems: a comprehensive survey. *Artificial Intelligence Review* 42, 4 (Dec. 2014), 767–799. https://doi.org/10.1007/s10462-012-9364-9

[27] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. 2011. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Rev.* 53, 2 (2011), 217–288. https://doi.org/10.1137/090771806 arXiv:https://doi.org/10.1137/090771806

[28] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (dec 2015), 19 pages. https://doi.org/10.1145/2827872

[29] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 173–182. https://doi.org/10.1145/3038912.3052569

[30] Balázs Hidasi and Alexandros Karatzoglou. 2018. Recurrent Neural Networks with Top-k Gains for Session-Based Recommendations. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Torino, Italy) *(CIKM '18)*. Association for Computing Machinery, New York, NY, USA, 843–852. https://doi.org/10.1145/3269206.3271761

[31] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE Computer Society, Los Alamitos, CA, USA, 263–272. https://doi.org/10.1109/ICDM.2008.22

[32] Nicolas Hug. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software* 5, 52 (Aug. 2020), 2174. https://doi.org/10.21105/joss.02174

[33] Dietmar Jannach and Gediminas Adomavicius. 2016. Recommendations with a Purpose. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. Association for Computing Machinery, New York, NY, USA, 7–10. https://doi.org/10.1145/2959100.2959186

[34] Dietmar Jannach and Christine Bauer. 2020. Escaping the mcnamara fallacy: towards more impactful recommender systems research. *AI Magazine* 41, 4 (2020), 79–95.

[35] O. Jeunen, K. Verstrepen, and B. Goethals. 2018. Fair Offline Evaluation Methodologies for Implicit-feedback Recommender Systems with MNAR Data. adrem.uantwerpen.be/bibrem/pubs/OfflineEvalJeunen2018.pdf. , 9 pages.

[36] Upulee Kanewala and James M. Bieman. 2018. Testing Scientific Software: A Systematic Literature Review. http://arxiv.org/abs/1804.01954

[37] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37. https://doi.org/10.1109/MC.2009.263

[38] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. 2004. pytest x.y. https://github.com/pytest-dev/pytest

[39] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–29. https://doi.org/10.1145/3360607

[40] Sara Latifi, Dietmar Jannach, and Andrés Ferraro. 2022. Sequential recommendation: A study on transformers, nearest neighbors and sampled metrics. *Information Sciences* 609 (Sept. 2022), 660–678. https://doi.org/10.1016/j.ins.2022.07.079

[41] Dung D. Le and Hady W. Lauw. 2017. Indexable Bayesian Personalized Ranking for Efficient Top-k Recommendation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM '17)*. Association for Computing Machinery, New York, NY, USA, 1389–1398. https://doi.org/10.1145/3132847.3132913

[42] Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman, and Tony Jebara. 2018. Variational Autoencoders for Collaborative Filtering. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 689–698.

https://doi.org/10.1145/3178876.3186150
[43] Nathan N. Liu, Min Zhao, Evan Xiang, and Qiang Yang. 2010. Online Evolutionary Collaborative Filtering. In *Proceedings of the Fourth ACM Conference on Recommender Systems* (Barcelona, Spain) *(RecSys '10)*. Association for Computing Machinery, New York, NY, USA, 95–102. https://doi.org/10.1145/1864708.1864729
[44] Malte Ludewig and Dietmar Jannach. 2018. Evaluation of Session-Based Recommendation Algorithms. *User Modeling and User-Adapted Interaction* 28, 4–5 (dec 2018), 331–390. https://doi.org/10.1007/s11257-018-9209-6
[45] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2019. Towards Deep Learning Models Resistant to Adversarial Attacks. https://doi.org/10.48550/arXiv.1706.06083
[46] William M McKeeman. 1998. Differential Testing for Software. 10, 1 (1998), 8.
[47] Lien Michiels, Robin Verachtert, and Bart Goethals. 2022. RecPack: An(other) Experimentation Toolkit for Top-N Recommendation using Implicit Feedback Data. In *Sixteenth ACM Conference on Recommender Systems*. ACM, Seattle WA USA, 648–651. https://doi.org/10.1145/3523227.3551472
[48] Andriy Mnih and Russ R Salakhutdinov. 2007. Probabilistic Matrix Factorization. In *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis (Eds.), Vol. 20. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2007/file/d7322ed717dedf1eb4e6e52a37ea7bcd-Paper.pdf
[49] Bamshad Mobasher, Robin Burke, Runa Bhaumik, and Chad Williams. 2007. Toward trustworthy recommender systems: An analysis of attack models and algorithm robustness. *ACM Transactions on Internet Technology* 7, 4 (Oct. 2007), 23–es. https://doi.org/10.1145/1278366.1278372
[50] Glenford J. Myers, Tom Badgett, and Corey Sandler. 2012. The Psychology and Economics of Software Testing. In *The Art of Software Testing* (1 ed.), Glenford J. Myers, Tom Badgett, and Corey Sandler (Eds.). Wiley, 5–18. https://doi.org/10.1002/9781119202486.ch2
[51] Glenford J. Myers, Tom Badgett, and Corey Sandler. 2012. Test-Case Design. In *The Art of Software Testing* (1 ed.), Glenford J. Myers, Tom Badgett, and Corey Sandler (Eds.). Wiley, 41–84. https://doi.org/10.1002/9781119202486.ch4
[52] Xia Ning and George Karypis. 2011. SLIM: Sparse Linear Methods for Top-N Recommender Systems. In *2011 IEEE 11th International Conference on Data Mining*. 497–506. https://doi.org/10.1109/ICDM.2011.134
[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[55] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *Commun. ACM* 62, 11 (oct 2019), 137–145. https://doi.org/10.1145/3361566
[56] Massimo Quadrana, Paolo Cremonesi, and Dietmar Jannach. 2018. Sequence-Aware Recommender Systems. *ACM Comput. Surv.* 51, 4, Article 66 (jul 2018), 36 pages. https://doi.org/10.1145/3190616
[57] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence* (Montreal, Quebec, Canada) *(UAI '09)*. AUAI Press, Arlington, Virginia, USA, 452–461.
[58] Steffen Rendle, Walid Krichene, Li Zhang, and Yehuda Koren. 2022. Revisiting the Performance of iALS on Item Recommendation Benchmarks. In *Sixteenth ACM Conference on Recommender Systems*. ACM, Seattle WA USA, 427–435. https://doi.org/10.1145/3523227.3548486
[59] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4902–4912. https://doi.org/10.18653/v1/2020.acl-main.442
[60] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering* 25, 6 (Nov. 2020), 5193–5254. https://doi.org/10.1007/s10664-020-09881-0
[61] Mohammad Saberian and Justin Basilico. 2021. RecSysOps: Best Practices for Operating a Large-Scale Recommender System. In *Fifteenth ACM Conference on Recommender Systems*. ACM, Amsterdam Netherlands, 590–591. https://doi.org/10.1145/3460231.3474620
[62] Aghiles Salah, Nicoleta Rogovschi, and Mohamed Nadif. 2015. A Dynamic Collaborative Filtering System via a Weighted Clustering approach. *Neurocomputing* 175 (10 2015). https://doi.org/10.1016/j.neucom.2015.10.050
[63] Aghiles Salah, Quoc-Tuan Truong, and Hady W Lauw. 2020. Cornac: A Comparative Framework for Multimodal Recommender Systems. *Journal of Machine Learning Research* 21, 95 (2020), 1–5.

[64] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (Sept. 2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[65] Ilya Shenbin, Anton Alekseev, Elena Tutubalina, Valentin Malykh, and Sergey I. Nikolenko. 2020. RecVAE: A New Variational Autoencoder for Top-N Recommendations with Implicit Feedback. In *Proceedings of the 13th International Conference on Web Search and Data Mining* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 528–536. https://doi.org/10.1145/3336191.3371831

[66] Harald Steck. 2019. Embarrassingly Shallow Autoencoders for Sparse Data. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 3251–3257. https://doi.org/10.1145/3308558.3313710

[67] Ruoyu Sun, Dawei Li, Shiyu Liang, Tian Ding, and Rayadurgam Srikant. 2020. The Global Landscape of Neural Networks: An Overview. *IEEE Signal Processing Magazine* 37, 5 (2020), 95–108. https://doi.org/10.1109/MSP.2020.3004124

[68] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Testing Deep Neural Networks. https://doi.org/10.48550/arXiv.1803.04792

[69] Quoc-Tuan Truong, Aghiles Salah, and Hady W. Lauw. 2021. Bilateral Variational Autoencoder for Collaborative Filtering. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining* (Virtual Event, Israel) *(WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 292–300. https://doi.org/10.1145/3437963.3441759

[70] Sakshi Udeshi and Sudipta Chattopadhyay. 2019. Grammar Based Directed Testing of Machine Learning Systems. http://arxiv.org/abs/1902.10027

[71] UNESCO. 2021. Recommendation on the Ethics of Artificial Intelligence. https://unesdoc.unesco.org/ark:/48223/pf0000380455.

[72] Robin Verachtert, Lien Michiels, and Bart Goethals. 2022. Are We Forgetting Something? Correctly Evaluate a Recommender System With an Optimal Training Window. In *Proceedings of the Perspectives on the Evaluation of Recommender Systems Workshop 2022*. CEUR-WS.org, Seattle WA USA.

[73] Sanne Vrijenhoek, Gabriel Bénédict, Mateo Gutierrez Granada, Daan Odijk, and Maarten De Rijke. 2022. RADio – Rank-Aware Divergence Metrics to Measure Normative Diversity in News Recommendations. In *Proceedings of the 16th ACM Conference on Recommender Systems* (Seattle, WA, USA) *(RecSys '22)*. Association for Computing Machinery, New York, NY, USA, 208–219. https://doi.org/10.1145/3523227.3546780

[74] Shoujin Wang, Xiuzhen Zhang, Yan Wang, Huan Liu, and Francesco Ricci. 2022. Trustworthy Recommender Systems. https://doi.org/10.48550/ARXIV.2208.06265

[75] Markus Weimer, Alexandros Karatzoglou, and Alex Smola. 2008. Improving Maximum Margin Matrix Factorization. *Machine Learning* 72. https://doi.org/10.1007/978-3-540-87479-9_12

[76] Eva Zangerle and Christine Bauer. 2022. Evaluating Recommender Systems: Survey and Framework. *ACM Comput. Surv.* 55, 8, Article 170 (dec 2022), 38 pages. https://doi.org/10.1145/3556536

[77] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* 48, 1 (Jan. 2022), 1–36. https://doi.org/10.1109/TSE.2019.2962027

A ALGORITHM WRAPPER

```python
from lenskit import batch, util
from lenskit.algorithms import Recommender
from pandas import DataFrame
from scipy.sparse import csr_matrix
from recpack.algorithms import Algorithm
from recpack.matrix import InteractionMatrix

def create_lenskit_dataset(X: InteractionMatrix) -> DataFrame:
    """Translate an InteractionMatrix into a
    dataframe structured as expected by LensKit algorithms."""
    lenskit_data = X._df[['uid', 'iid', 'ts']].copy()
    lenskit_data["rating"] = 1
    lenskit_data.rename(
        columns={'uid': 'user', 'iid': 'item', 'ts': 'timestamp'}, inplace=True)
    lenskit_data = lenskit_data[['user', 'item', 'rating', 'timestamp']].copy()
    # Lenskit does not allow duplicates
    lenskit_data.drop_duplicates(
        subset=["user", "item"], keep="first", inplace=True)
    return lenskit_data

class LenskitWrapper(Algorithm):
    """Wrapper class to use LensKit algorithms in the RecPack framework.

    :param model_class: The LensKit algorithm class
    :type model_class: type
    :param kwargs: The arguments for the LensKit model, as keyword arguments
    """
    def __init__(self, model_class, **kwargs):
        fittable = model_class(**kwargs)
        fittable = util.clone(fittable)
        self.lenskit_model = Recommender.adapt(fittable)

    def _transform_fit_input(self, X: InteractionMatrix) -> csr_matrix:
        self._assert_is_interaction_matrix(X)
        return X

    def _transform_predict_input(self, X: InteractionMatrix) -> csr_matrix:
        self._assert_is_interaction_matrix(X)
        return X

    def _fit(self, X: InteractionMatrix):
        self.lenskit_model.fit(create_lenskit_dataset(X))
        self.fitted_ = True # for check_fitted

    def _predict(self, X: InteractionMatrix) -> csr_matrix:
        users = X.active_users
        recs = batch.recommend(self.lenskit_model, users, 100)
        values = recs.score.values
        indices = recs[["user", "item"]].values
        indices = indices[:, 0], indices[:, 1]

        matrix = csr_matrix((values, indices), dtype=float, shape=X.shape)

        return matrix
```

```python
class LensKitSimilarityAlgorithmWrapper(LenskitWrapper):
    """Specific wrapper for Similarity algorithms,
    translating the similarity matirx from the lenskit model
    to the expected RecPack structure
    """

    @property
    def similarity_matrix_(self):
        return self.lenskit_model.predictor.sim_matrix_.to_scipy()

class LensKitFactorizationAlgorithmWrapper(LenskitWrapper):
    """Specific wrapper for factorization algorithms,
    translating the embeddings from the lenskit model
    to the expected RecPack structure
    """

    @property
    def item_embedding_(self):
        return self.lenskit_model.predictor.item_features_.T
```

B DETAILED TEST RESULTS

	BBST1+	*BBST1-*	*BBST2*	*BBST3*	*WBST1Sim+*	*WBST1Sim-*	*WBST1Emb-*	*WBST1Emb+*
BPRMF	1✓	1✓	1✓	1×				
EASE	1✓	1✓	1✓		1✓	1✓		
GRU4RecCrossEntropy	1✓	1✓	1✓	1✓				
GRU4RecNegSampling	1✓	1✓	1✓	1✓				
ItemKNN	4✓	4✓	4✓		4✓	4✓		
MultVAE	1✓	1✓	1✓	1✓				
NMF	1✓	1✓	1✓	1✓			1✓	1✓
NMFItemToItem	1✓	1✓	1✓	1✓	1✓	1✓		
Popularity	1✓	1✓	1✓					
Prod2Vec	3✓	3✓	3✓	3✓	3✓	3✓		
Prod2VecClustered	1✓	1✓	1✓	1✓	1✓	1✓		
Random	1✓	1✓	1✓					
RecVAE	1✓	1✓	1✓	1✓				
SLIM	3✓	3✓	3✓		3✓	3✓		
STAN	1✓	1✓	1✓					
SVD	1✓	1✓	1✓	1✓			1✓	1✓
SVDItemToItem	1✓	1✓	1✓	1✓	1✓	1✓		
SequentialRules	1✓	1✓	1✓		1✓	1✓		
TARSItemKNN	2×	2✓	2✓		2×	2✓		
WMF	1✓	1✓	1✓					

Table 12. Part 1 of the detailed test results for RecPack. A ✓indicates the algorithm passed the test case, a × indicates it failed the test case. A blank space signifies the test case was not run for this algorithm. When an algorithm has hyperparameters that give rise to distinct code paths, multiple instances of the algorithm were tested with different values of the hyperparameter. The numbers indicate how many of these instances failed or passed a test, e.g., 2✓2× implies two instances passed and two failed.

	BBIT1TopK	*BBIT1Seq*	*BBIT2TopK-A*	*BBIT2TopK-B*	*BBIT2Seq-A*	*BBIT2Seq-B*	*BBIT3*	*BBIT4*	*BBIT5*	*BBIT6*
BPRMF	1×		1✓	1✓			1✓	1×	1✓	1×
EASE	1✓		1✓	1✓			1✓	1✓	1✓	1✓
GRU4RecCrossEntropy		1×			1×	1×	1✓	1✓	1✓	1×
GRU4RecNegSampling		1×			1×	1×	1✓	1✓	1✓	1×
ItemKNN	4✓		4✓	4✓			4✓	4✓	4✓	4✓
MultVAE	1✓		1✓	1✓			1✓	1✓	1✓	1×
NMF	1✓		1×	1×			1×	1×	1✓	1✓
NMFItemToItem	1✓		1✓	1✓			1✓	1✓	1✓	1✓
Popularity							1✓	1×	1✓	1✓
Prod2Vec	3✓		3✓	3✓			3✓	3✓	3✓	3×
Prod2VecClustered			1×	1✓			1✓	1✓	1✓	1×
Random							1✓	1✓		
RecVAE	1✓		1×	1×			1✓	1✓	1✓	1×
SLIM	3✓		3✓	3✓			3✓	3✓	3✓	3×
STAN		1×			1✓	1✓	1✓	1✓	1✓	1✓
SVD	1✓		1✓	1✓			1×	1×	1✓	1×
SVDItemToItem	1✓		1✓	1✓			1✓	1✓	1✓	1✓
SequentialRules		1✓			1✓	1✓	1✓	1✓	1✓	1✓
TARSItemKNN			2✓	2✓			2✓	2✓	2✓	2✓
WMF	1✓		1✓	1✓			1✓	1✓	1✓	1×

Table 12. Part 2 of the detailed test results for RecPack. A ✓indicates the algorithm passed the test case, a × indicates it failed the test case. A blank space signifies the test case was not run for this algorithm. When an algorithm has hyperparameters that give rise to distinct code paths, multiple instances of the algorithm were tested with different values of the hyperparameter. The numbers indicate how many of these instances failed or passed a test, e.g., 2✓2× implies two instances passed and two failed.

	WBIT1Emb	*WBIT1Sim*	*WBIT2Sim*	*WBUT1Iter+*	*WBUT1Iter-*
BPRMF				1✓	1✓
EASE		1✓	1✓		
GRU4RecCrossEntropy				1✓	1✓
GRU4RecNegSampling				1✓	1✓
ItemKNN		4✓	4✓		
MultVAE				1✓	1✓
NMF	1✓				
NMFItemToItem		1✓	1✓		
Popularity					
Prod2Vec		3×	3✓	3✓	3✓
Prod2VecClustered		1×	1✓	1✓	1✓
Random					
RecVAE				1✓	1✓
SLIM		3✓	3✓		
STAN					
SVD	1✓				
SVDItemToItem		1✓	1✓		
SequentialRules		1×	1✓		
TARSItemKNN		2✓	2✓		
WMF					

Table 12. Part 3 of the detailed test results for RecPack. A ✓indicates the algorithm passed the test case, a × indicates it failed the test case. A blank space signifies the test case was not run for this algorithm. When an algorithm has hyperparameters that give rise to distinct code paths, multiple instances of the algorithm were tested with different values of the hyperparameter. The numbers indicate how many of these instances failed or passed a test, e.g., 2✓2× implies two instances passed and two failed.

	ImplicitMF	ItemKNN	Popular	UserKNN
BBST1+	1✓	1✓	1✓	1✓
BBST1-	1✓	1✓	1✓	1✓
BBST2[c]	1✓	1✓	1✓	1✓
BBST3	1✓			
WBST1Sim+		1✓		
WBST1Sim-		1×		
WBST1Emb+	1✓			
WBST1Emb-	1×			
BBIT1TopK	1✓	1✓		1✓
BBIT2TopK-A	1✓	1✓		1✓
BBIT2TopK-B	1✓	1✓		1✓
BBIT5	1✓	1✓	1✓	1✓
BBIT6	1×	1✓	1✓	1✓
WBIT1Emb	1✓			
WBIT1Sim		1✓		
WBIT2Sim		1✓		

Table 13. Detailed test results for `PyLensKit`. A ✓indicates the algorithm passed the test case, a × indicates it failed the test case. A blank space signifies the test case was not run for this algorithm. When an algorithm has hyperparameters that give rise to distinct code paths, multiple instances of the algorithm were tested with different values of the hyperparameter. The numbers indicate how many of these instances failed or passed a test, e.g., 2✓2× implies two instances passed and two failed.

	CoClustering	ItemKNN	UserKNN	NMF	SVD	SVDpp
BBST1+	1✓	4✓	4✓	1✓	1✓	1✓
BBST1-	1×	3✓1×	1✓3×	1✓	1×	1×
BBST2	1✓	4✓	4✓	1✓	1✓	1✓
BBST3	1×			1✓	1×	1×
WBST1Sim+		3✓1×				
WBST1Sim-		4×				
WBST1Emb+				1✓	1✓	1✓
WBST1Emb-				1✓	1×	1×
BBIT1TopK	1✓	4✓	4✓	1✓	1×	1×
BBIT2TopK-A	1×	4✓	3✓1×	1×	1✓	1✓
BBIT2TopK-B	1×	4✓	4✓	1×	1✓	1✓
BBIT5	1✓	4✓	4✓	1✓	1✓	1✓
BBIT6	1✓	4✓	4✓	1✓	1✓	1✓
WBIT1Emb				1✓	1×	1×
WBIT1Sim		4✓				
WBIT2Sim		4×				

Table 14. Detailed test results for `Surprise`. A ✓indicates the algorithm passed the test case, a × indicates it failed the test case. A blank space signifies the test case was not run for this algorithm. When an algorithm has hyperparameters that give rise to distinct code paths, multiple instances of the algorithm were tested with different values of the hyperparameter. The numbers indicate how many of these instances failed or passed a test, e.g., 2✓2× implies two instances passed and two failed.

	BBST1+	*BBST1-*	*BBST2*	*BBST3*	*BBIT1TopK*	*BBIT2TopK-A*	*BBIT2TopK-B*	*BBIT5*	*BBIT6*
BPR	1✓	1✓	1✓	1✓	1×	1×	1×	1✓	1✓
BiVAECF	8✓	8✓	8✓	4✓4×	6✓2×	4✓4×	5✓3×	8✓	8✓
GMF	4✓	4✓	4✓	2✓2×	4×	4✓	4✓	4✓	4×
IBPR	1✓	1✓	1✓		1×	1✓	1✓	1✓	1×
ItemKNN	1✓	1✓	1✓		1✓	1✓	1✓	1✓	1✓
MF	4✓	4✓	4✓	4×	4×	4✓	4✓	4✓	4✓
MLP	9✓	9✓	9✓	7✓2×	9×	9✓	9✓	9✓	9×
MMMF	1✓	1✓	1✓	1✓	1×	1✓	1✓	1✓	1✓
MostPop	1✓	1✓	1✓		1×			1✓	1✓
NMF	2✓	2✓	2✓	2×	2×	2✓	2✓	2✓	2✓
NeuMF	10✓	10✓	10✓	8✓2×	10×	10✓	10✓	10✓	10×
PMF	2✓	2✓	2✓	1✓1×	1✓1×	2✓	2✓	2✓	2✓
SKMeans	1✓	1✓	1✓	1✓	1✓	1✓	1✓	1✓	1✓
UserKNN	1✓	1✓	1✓		1✓	1✓	1✓	1✓	1✓
VAECF	8✓	8✓	8✓	8✓	3✓5×	4✓4×	3✓5×	8✓	8✓
WBPR	1✓	1✓	1✓	1✓	1×	1✓	1✓	1✓	1✓
WMF	1✓	1✓	1✓	1✓	1×	1✓	1✓	1✓	1✓

Table 15. Detailed test results for `Cornac`. A ✓indicates the algorithm passed the test case, a × indicates it failed the test case. A blank space signifies the test case was not run for this algorithm. When an algorithm has hyperparameters that give rise to distinct code paths, multiple instances of the algorithm were tested with different values of the hyperparameter. The numbers indicate how many of these instances failed or passed a test, e.g., 2✓2× implies two instances passed and two failed.