sustainability

Article

Automatic Grading Tool for Jupyter Notebooks in Artificial Intelligence Courses

Cristian D. González-Carrillo [1,*,†], Felipe Restrepo-Calle [1], Jhon J. Ramírez-Echeverry [2] and Fabio A. González [1]

[1] Department of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá 111321, Colombia; ferestrepoca@unal.edu.co (F.R.-C.); fagonzalezo@unal.edu.co (F.A.G.)
[2] Department of Electrical and Electronics Engineering, Universidad Nacional de Colombia, Bogotá 111321, Colombia; jjramireze@unal.edu.co
* Correspondence: crdgonzalezca@unal.edu.co
† Current address: Google, Mountain View, CA 94043, USA.

Abstract: Jupyter notebooks provide an interactive programming environment that allows writing code, text, equations, and multimedia resources. They are widely used as a teaching support tool in computer science and engineering courses. However, manual grading programming assignments in Jupyter notebooks is a challenging task, thus using an automatic grader becomes a must. This paper presents UNCode notebook auto-grader, that offers summative and formative feedback instantaneously. It provides instructors with an easy-to-use grader generator within the platform, without having to deploy a new server. Additionally, we report the experience of employing this tool in two artificial intelligence courses: *Introduction to Intelligent Systems* and *Machine Learning*. Several programming activities were carried out using the proposed tool. Analysis of students' interactions with the tool and the students' perceptions are presented. Results showed that the tool was widely used to evaluate their tasks, as a large number of submissions were performed. Students expressed positive opinions mostly, giving feedback about the auto-grader, highlighting the usefulness of the immediate feedback and the grading code, among other aspects that helped them to solve the activities. Results remarked on the importance of providing clear grading code and formative feedback to help the students to identify errors and correct them.

Keywords: auto-grading systems; jupyter notebooks; artificial intelligence; computer programming; formative feedback; summative feedback; assessment; sustainable development

check for updates

Citation: González-Carrillo, C.D.; Restrepo-Calle, F.; Ramírez-Echeverry, J.J.; González, F.A. Automatic Grading Tool for Jupyter Notebooks in Artificial Intelligence Courses. *Sustainability* **2021**, *13*, 12050. https://doi.org/10.3390/su132112050

Academic Editors: Danial Hooshyar, Michael D. Kickmeier-Rust and Nour El Mawas

Received: 28 August 2021
Accepted: 26 October 2021
Published: 31 October 2021

1. Introduction

The Jupyter notebook (previously known as IPython Notebook) is an open-source tool where users have an interactive programming environment for scientific computing, which allows writing code, text, equations, and multimedia resources [1]. Due to its increasing usage, it has become the preferred computational notebook in many areas, such as: machine learning, artificial intelligence, data science, among others [2,3]. Jupyter notebooks have been recently used as a teaching and learning supporting tool in academic institutions, such as: Universidad Complutense de Madrid [4], BVB College of Engineering and Technology [5] and at the University of Illinois at Urbana-Champaign [6]. They have been used for teaching in different areas like radiology [7], geography [8], geology [9], among others [10]. Jupyter notebooks have also had a remarkable adoption in engineering and computer science courses [6,11]. In addition to their convenience as an interactive handout tool, Jupyter notebooks are used as a mechanism for assigning and collecting homework [12]. Usually, instructors provide Jupyter Notebooks templates to the students to guide them towards the expected solution, like in the course *Introduction to Data Science* at the University of Illinois at Urbana-Champaign [6]. However, assessing and grading programming activities in a manual way is a tedious and time-consuming task [13], which may lead to assessment errors and biased grades [14].

Automatic grading tools for programming activities have been mostly used for competitive programming competitions, like the ICPC (International Collegiate Programming Contest | https://icpc.global, accessed on: 29 October 2021) or *Codeforces* (http://codeforces.com, accessed on: 29 October 2021). For teaching purposes, many automatic grading tools have been developed, such as: Check50 [15], Flexible Dynamic Analyzer (FDA) [16], among others [17]. However, these tools do not provide automatic grading capabilities for Jupyter notebooks. As a response to the lack of automatic grading tools for Jupyter Notebooks, *nbgrader* has been developed by the Jupyter team since 2014 [18]. It has been widely used by the academic community; by 2018, more than 10,000 notebooks using *nbgrader* were located on GitHub and implemented in several universities. For instance, it is used in [11,19,20]. Nevertheless, using *nbgrader* requires deploying the JupyterHub server (https://jupyter.org/hub, accessed on: 29 October 2021) to properly deliver assignments and collect submissions. Other approaches for automatic grading of notebook-based activities include: *OK* (https://okpy.org, accessed on: 29 October 2021), which has been used in Sridhara et al. [21]; *Otter-Grader* (https://otter-grader.readthedocs.io, accessed on: 29 October 2021); *Web-CAT* [22]; and an extended version of *check50* [15]. Moreover, there are some popular commercial applications providing automatic grading in online learning platforms, such as: *Coursera* (https://www.coursera.org, accessed on: 29 October 2021) and *Vocareum* (https://www.vocareum.com, accessed on: 29 October 2021).

Improving the quality of education is one of the goals of sustainable development. As shown in the works discussed above, automatic grading tools are important assets to improve the quality of education [23]. They help to provide timely feedback to students during the learning process, help to expand the education system coverage and support the development of systems for autonomous lifelong learning. Many of these systems provide feedback to students with a partial grade, through several ways to collect submissions, having different options to configure and create the assignments. However, various opportunities arise to tackle different limitations of these tools, such as: the assignment configuration can be tedious, as in some cases it must be done manually; in most cases, no graphical user interfaces (GUI) are provided to create tests cases; and more importantly, the feedback given to students on failed submissions does not provide sufficiently detailed information to allow learners not only to find their errors but also to know how to proceed to correct them (i.e., formative feedback [24]).

The first objective of this paper is to introduce the *UNCode notebook auto-grading tool*, where students can obtain detailed and formative feedback instantaneously. The novel tool was built on top of UNCode [25], which is an educational environment to automatically grade traditional programming activities in introductory programming courses. The second objective is to report the experience of using the UNCode Jupyter notebook auto-grader in two Artificial Intelligence (AI) courses: *Introduction to Intelligent Systems* and *Machine Learning*. For this purpose, we have analyzed quantitative data from the assignments graded using the tool and also qualitative data related to a survey conducted among students regarding their perception about the tool. This effort is motivated by two main research questions (RQ):

- RQ1: *How do students interact with the UNCode notebook auto-grader?*
- RQ2: *What is the students' perception of the tool as a support mechanism for their learning process?*

It is worth clarifying what we understand with interaction and perception in this study: interaction refers to the way the students use the UNCode notebook auto-grader, that is, the students' submissions per activity and the category of the obtained feedback. By perception we refer as how the students assess and judge the utility of the tool and the different features in regard of their learning process, this measured by labeling the students' answers to open-ended questions regarding this topic. This work contributes to the area of computer science and engineering education in two ways: first, with the proposed self-grading tool itself, which provides instructors with an easy-to-use automatic grader that offers formative feedback to students and its source code is publicly available; second, with the report of the experience of using the tool in an academic context, which offers insights to better understand the potential benefits of the tool for the students' learning process.

The remainder of this paper is organized as follows: Section 2 describes related works on automatic grading tools for Jupyter Notebooks and feedback mechanisms on tools that support the students' learning process. Section 3 presents the UNCode notebook auto-grader and its technical specifications. Next, Section 4 details the materials and methods, including descriptions of the AI courses, participants, instruments, procedures, and data analysis. Section 5 reports the results of the collected data with respect to the students' interaction with the UNCode notebook auto-grading tool and the conducted survey. Section 6 discusses the obtained results, and finally, Section 7 concludes the paper with final remarks and describes future directions of research.

2. Background and Related Works

Jupyter Notebooks have become the preferred computational notebook in many computer science areas [2,3,6], as well as in different science areas and courses [7–10]. Therefore, they are used as a teaching and learning supporting tool in academic institutions [4,5]. For teaching purposes, many automatic grading tools have been developed like [15–17], but these tools do not provide automatic grading capabilities for Jupyter notebooks. In that context, this section is split in two parts: the first one is focused on the related works and tools that provide Jupyter notebook auto-grading capabilities; the second part is intended to give a background on the kinds of feedback provided by software tools designed to support students in their learning process of courses related to computer programming.

2.1. Related Works: Automatic Grading Tools for Jupyter Notebooks

There are several ways to use Jupyter notebooks for teaching and learning in a course, as it was explained in Barba et al., 2019 [1], especially in computer science courses. When they are used during the assessment process, having an automatic grading tool for Jupyter notebook assignments is highly recommended. *Nbgrader* [18] was built in 2014 to begin addressing this need. It is a grading tool that provides an easy-to-use environment for creating, delivering, collecting, and grading submissions, as well as widely used in several systems [11,19,20]. It can be used on the instructor's local computer or on a JupyterHub server, although, in the first case the workflow is more limited; for instance, it is not possible to collect submissions. On the other hand, when a JupyterHub server is used, all grading functionalities are unlocked and the grading process is easier, i.e., students submit their solutions within the same platform; however, it requires a centralized server-based installation, which consumes more computational resources.

Several commercial platforms that also support Jupyter notebook automatic graders have been developed. Some of these are: *Vocareum* and *CoCalc* (https://cocalc.com, accessed on: 29 October 2021). *Vocareum* provides a cloud-based fully hosted solution for Jupyter notebooks with automatic grading via nbgrader. This offers test generation through a user interface, generating automated scoring and inline code feedback. This platform is quite similar to *CoCalc*, supporting almost the same features. As they are commercial applications, it is necessary to pay for their services. Due to this, it can be expensive to host several or large courses. In addition, another commercial platform that offers Jupyter notebook auto-grading is *Coursera*, whose automatic grading system supports several Massive Open Online Courses (MOOCs) within the platform, e.g., the *Deep Learning Specialization* (https://www.coursera.org/specializations/deep-learning, accessed on: 29 October 2021). Here, students can solve and submit assignments in a cloud-based Jupyter environment. Nevertheless, students can only see whether the test has passed or failed; it does not provide additional information like comparing answers or runtime errors.

Manzoor et al. [22] have added support for autograding Jupyter notebooks in their already working automatic grading system called *Web-CAT*; they use nbgrader with some additional modifications. They have made efforts on delivering immediate feedback to students using *nbgrader*, which can be also used via a Learning Management System (LMS) to upload the submission, and reported good results when students and instructors used it.

Otter-Grader offers a Command Line Interface (CLI) and Application Programming Interface (API) in Python to instructors to configure the assignments and run submissions either in the instructor's machine, in a sandboxed environment, or using *Gradescope* [26]. However, this process requires instructors to manually configure the assignments. In addition, in case it is graded locally, the delivery and collection of assignment demands using another platform, as well as the feedback is not given to students instantly.

OK is a widely used grading system at UC Berkeley, supporting Jupyter notebook automatic grading in Data Science courses. The way the notebooks are graded is via configuration files, which contain Python code in doctest format (https://docs.python.org/3/library/doctest.html, accessed on: 29 October 2021). This eases the delivery, submission, and grading process for students and instructors via an API and a CLI. However, the test creation process is not straightforward, as they must be created manually, requiring users to have additional knowledge about this. Moreover, with respect to feedback, it has some limitations when the submission fails as expressed by Sharp et al. [15].

Table 1 presents a comparison of the aforementioned related automatic grading tools for Jupyter notebooks. We included the tool presented in this work, UNCode Notebook, as the last column of the table. The following are the comparison criteria: *JupyterHub not required* to determine if the tool requires to deploy this additional server to deliver and collect submissions; *Sandbox* to protect and isolate grading code; *Cost free*; *In-platform automatic test generation* whether the tool supports the automatic generation of tests; whether the tool provides *Immediate feedback*; and the support of *In-platform submission collection*. The comparison criteria were selected from the different relevant features these systems provide.

Table 1. Comparison of auto-grading tools for Jupyter notebooks including: Nbgrader, Web-CAT, Otter-Grader, OK, Coursera, Vocareum, CoCalc and UNCode notebook (this work).

Category	**Nbgrader**	**Web-CAT**	**Otter-Grader**	**OK**	**Coursera**	**Vocareum**	**CoCalc**	**UNCode Notebook**
JupyterHub not required	✓ †	✗	✓	✓	✗	✗	✗	✓
Sandbox	✓ †	○	✓	○	○	○	○	✓
Cost free	✓	✓	✓	✓	✗	✗	✗	✓
In-platform automatic test generation	✓	✓	✗	○	✗	✓	✓	✓
Immediate feedback	✓ †	✓	✓ †	✓	✓ ‡	✓	✓	✓
In-platform assignment collection	✓ †	✓	✓ †	✓	✓	✓	✓	✓

† Not provided by default, it is configurable; ‡ Limited; ○ Information not available.

From Table 1 and the above discussed tools, it can be seen that most tools depend on JupyterHub server to extend the grading functionalities. Though, this might be a limitation in many cases as deploying this server requires more computational resources. Moreover, some systems are not free or publicly available, which may imply high costs for several courses. In addition, a few tools do not provide user interfaces to generate tests. Furthermore, users must be able to easily create, deliver, and submit assignments, and therefore, receive instant feedback. Taking that into consideration, we decided to develop a new Jupyter Notebook auto-grader tool, drawing inspiration from the advantages of the related works. The new tool, UNCode Notebook, provides the following features: automatic test generation within the same platform, a sandboxed environment, immediate and formative feedback, and a platform where assignments are easily delivered and collected.

2.2. Related Works: Feedback on Tools to Support the Learning Process

Many software tools have been proposed to support students in the learning process of computer programming. According to Keuning et al. [27], these tools offer feedback to students in several forms that can be grouped into two main categories as proposed by Narciss [24]: knowledge about mistakes and knowledge about how to proceed. In the context of assessment tools, these categories of feedback can be associated with *summative* and *formative feedback*, respectively. Most measures and metrics obtained from feedback that provide knowledge about mistakes can be associated with summative feedback, such as feedback categories and grades, whereas "*information communicated to the learner that is intended to modify their or her thinking or behavior for the purpose of improving learning*" [28] can be understood as formative feedback.

Several evaluation tools provide summative feedback in the form of grades or percentages for assessments in programming assignments, which can be primarily useful in identifying mistakes [27]. According to Gordillo [29], incorporating automated assessment tools into courses is beneficial to students because it increases their motivation, improves the quality of their programs, and enhances their practical programming skills. However, a non-negligible percentage of students often find the feedback provided insufficient; in some cases, they find it difficult to understand, and the grades may not be fair. Therefore, as Keuning et al. [27] state, the information that a grade provides about the similarities and differences between the appropriate standards for a given task and the characteristics of the student's work is often only superficial. In most cases, feedback is binary in nature (pass/fail) [30]. To improve feedback, researchers have proposed different alternatives, such as a methodological approach that combines automatic assessment with human reviews in a course with a large number of students supported by a small number of teachers [31], and a novel tool in which students can choose to "buy" hints and solutions [32].

To enrich the summative feedback of tools that support programming learning, recent works have explored a wide spectrum of possibilities. However, these innovative proposals do not support automatic evaluation processes in most cases. Some of the more relevant examples to support programming learning include: a study of different mechanisms for compiling and presenting error messages [33]; a tool to help novice programmers improve their understanding of the debugging process and enhance their debugging skills [34]; a tool that provides an interactive visualization of the evolution of the student's code throughout the completion of a task, facilitating discussion of the intermediate steps, and not just a single final submission [35]; a tool that offers a formative assessment automatically as students program chatbots using fundamental programming constructs [36]; and a methodological approach involving peer code review activities to help students understand programming concepts and improve their programming skills [37].

Furthermore, there is a growing need for assessment tools that provide learners with formative feedback [27]. According to Loksa et al. [38], designers of introductory programming learning technologies could incorporate explicit instructions on the stages of problem solving, find ways to detect what stage a learner is at, and provide constructive feedback on the strategies and tactics to proceed, i.e., metacognitive awareness. Similarly, Prather et al. [30] suggest that automated assessment tools should be modified to provide a more comprehensive cognitive scaffolding around which students can appropriately place their knowledge as they learn. In addition, Ullah et al. [39] recommend providing complete and instantaneous information about the location of student errors and suggesting solutions or guidelines for resolving them. In other words, recent research points to the need for formative feedback that is offered to students in assessment tools, supporting students during their learning process and providing them with the appropriate functionality and information to build knowledge on how to proceed in case of errors.

Consequently, more research is needed to design evaluation tools that not only support summative evaluation, but also provide some form of formative feedback during evaluation, and at the same time, there is a need to better understand the benefits and challenges of using these tools to support evaluation in computer science and engineering courses. Therefore, the presented automatic grading tool for Jupyter notebooks in this paper provides to the students immediate summative and formative feedback. For instance, it immediately returns the grade for the activity as summative feedback and supplies additional information like runtime errors and output difference with respect to the expected program output as formative feedback.

3. UNCode Notebook Auto-Grader

To develop UNCode notebook auto-grader, we add the support for the automatic evaluation of Jupyter notebooks to UNCode. It is an open source educational environment to automatically grade programming assignments [25], which is built on top of *INGInious* [40]. UNCode is used and maintained by the Universidad Nacional de Colombia, and is deployed at https://uncode.unal.edu.co (accessed on: 29 October 2021). UNCode facilitates the creation and evaluation of programming activities and provides instantaneous formative feedback [41]. The supported programming languages are Java, Python, C/C++. Additionally, it supports hardware description languages (HDLs) like Verilog and VHDL. In general, instructors automatically create and deliver assignments to students within the same platform with the help of a graphical user interface to ease the process. Moreover, students access the assignment and solve it following the given specifications, submitting the source code of their solutions. The source code is tested using a variety of test cases, and the students instantly receive feedback, highlighting the failed tests with the associated informative error, as well as the corresponding grade (more information at: https://juezun.github.io, accessed on: 29 October 2021).

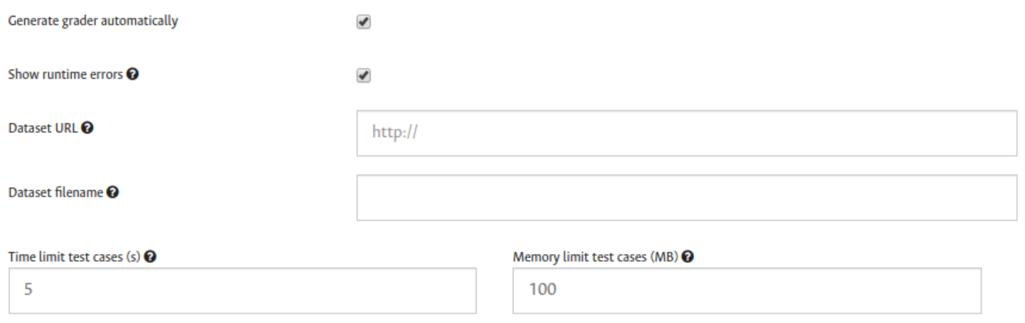
In this context, we developed support to automatically grade Jupyter notebooks on top of UNCode. This tool is called UNCode notebook auto-grader, and it is publicly available under the GNU Affero General Public License v3.0 (AGPL-3.0) (GitHub repository | https://github.com/JuezUN, accessed on: 29 October 2021). That way, it is not necessary to deploy a new service, and we can take advantage of UNCode for grading Jupyter notebooks. To accomplish this, an easy-to-use user interface was developed for instructors to configure the Jupyter notebook grader. They can create different grading tests using Python code and generate automatically the configuration files. Then, the assignments are delivered to the students on UNCode, where they can either download the students' version of the notebook and solve it locally in the student's computer, or open the student's version in a cloud-based Jupyter notebook execution environment like *Google Colaboratory* (Colab | https://colab.research.google.com, accessed on: 29 October 2021). After solving the assignment, students submit their solution to UNCode and they receive instant feedback along with the grade corresponding to the submission results. This submission is executed in a secure sandbox using *Docker* (https://www.docker.com, accessed on: 29 October 2021), which is in charge of running the submission with the provided tests, with the help of *OK CLI*, and the corresponding feedback is generated. We decided to use *OK CLI* because this open source tool already has a strong testing system where several test cases can be created, and it provides automatic summative feedback, which is helpful to grade Jupyter notebooks. Therefore, we added complementary information to provide additional formative feedback within UNCode notebook auto-grader.

The presented feedback is marked with different categories: *Accepted* when the cases are correct, *Wrong Answer* in case the output is not the expected one, *Runtime Error* is shown when student's code throws an exception, *Grading Runtime Error* is an exception raised by the grading code. In case the submission takes too long, it is categorized as *Time Limit*, and *Memory Limit* when it exceeds the allowed memory. In that context, the process can be divided in two big stages: the first stage corresponds to the development of the user interface for instructors; the second stage is related to the execution of the submission and feedback generation. These stages are explained below.

3.1. Automatic Assignment Configuration

The first stage involved developing a user interface for instructors to create and configure the assignments on UNCode. To correctly configure the automatic grader, the student's notebook version must have been already created, with this, the instructor will be able to create different tests. To start, the instructor has to create a *task* on UNCode and fill in the task settings. Figure 1 shows some initial general settings for the grader, as explained below:

- *Generate grader automatically*: this is left as optional since the user might want to automatically generate the grader or not. This option is checked by default.
- *Show runtime errors* option is set to choose whether or not to show runtime errors to students when the feedback is given. In case this is checked, additional feedback is shown, such as: *Runtime* or *Memory Limit Errors*. Otherwise, the student will receive only *Accepted* or *Wrong Answer* results as the feedback.
- *Dataset URL* and *Dataset filename* options are given in case the assignment requires a dataset. To set an eventual dataset for the automatic grader, there are two options: first, add the dataset URL and filename, informing the grader that a dataset must be downloaded right before each submission is executed. The second option consists of uploading the dataset directly to the task file system.
- *Grading limits*: here the instructor is able to set time and memory limits for each test. This will determine when *Time limit* or *Memory limit* errors are shown within the feedback.



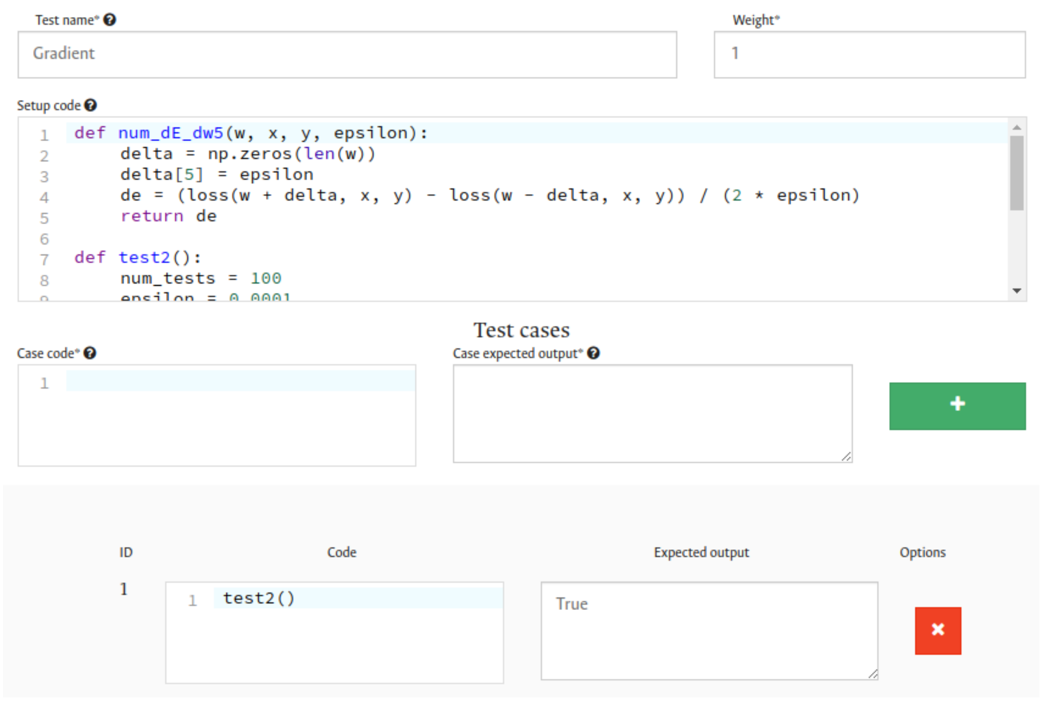
Generate grader automatically	✓
Show runtime errors	✓
Dataset URL	http://
Dataset filename	
Time limit test cases (s)	5
Memory limit test cases (MB)	100

Figure 1. Initial general settings for the Jupyter notebook grader configuration.

After these initial general settings, the grading code can be added. The grader is composed of a group of *tests*, which aims to grade or evaluate a specific functionality of the assignment. For that, a *test* is divided into *test cases*, which grade the target functionality with more granularity. For instance, the student is asked to implement a function that sums two numbers and returns the result, then, a *test* groups a series of *test cases*, each test case will then evaluate the function through different parameters. This to finally determine whether the implemented functionality is correct or not. The instructor is free to choose a way the tests and test cases are created and how they evaluate a certain code, all of that depending on the assignment's goals.

In that context, to create a test, a modal window is displayed with all fields to be filled in as seen in Figure 2. The different options for the test configuration are described as follows:

- *Test name*: this is used to identify this test among all other tests.
- *Weight*: used to determine the importance of this test among the others when the grade is calculated.
- *Setup code*: code used across all test cases in this specific test. It is intended to help the instructor to reuse the code and facilitate the test creation process when there are several test cases.
- *Test cases section*: the instructor must add all necessary test cases. A test case is composed of the test code and an expected output, which determines the correctness of each test case.



Test name*	Weight*
Gradient	1

Setup code

```python
def num_dE_dw5(w, x, y, epsilon):
    delta = np.zeros(len(w))
    delta[5] = epsilon
    de = (loss(w + delta, x, y) - loss(w - delta, x, y)) / (2 * epsilon)
    return de

def test2():
    num_tests = 100
    epsilon = 0.0001
```

Test cases

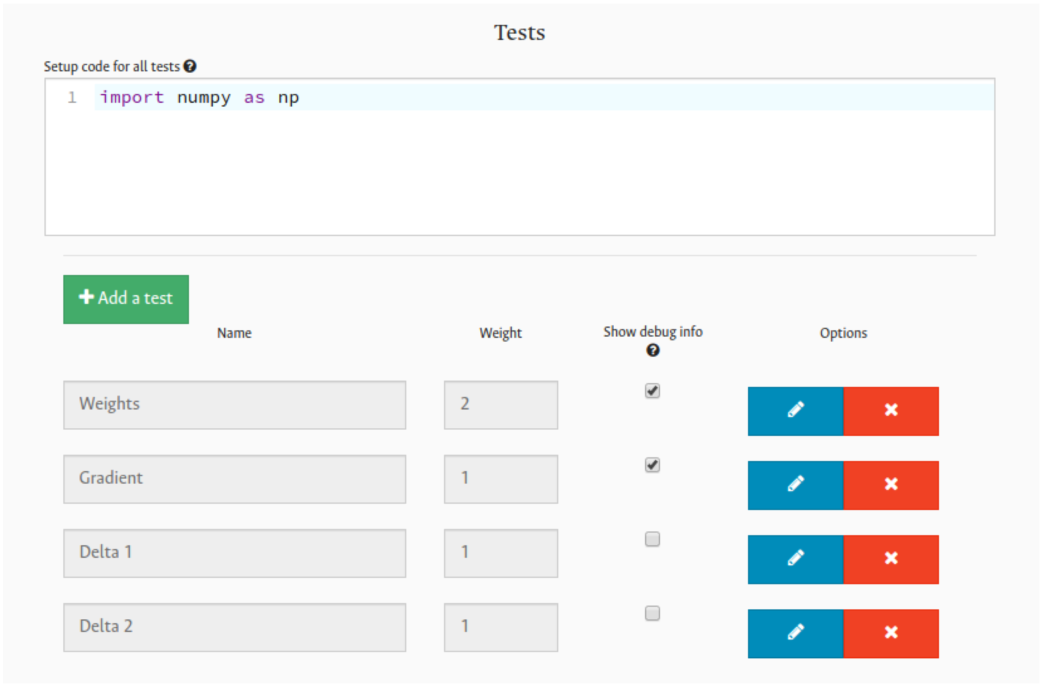
Case code*

Case expected output*

ID	Code	Expected output	Options
1	test2()	True	✖

Figure 2. Screenshot showing the form to create or modify a test and its corresponding test cases.

Once the test is saved, it will be added to the *tests section* in the grader configuration as shown in Figure 3. Here, there are four tests that have already been created to evaluate the assignment. Furthermore, these tests can be deleted or edited. Additionally, the instructor may choose which test will show additional information within the feedback to students. This feature aims to help students to debug their code when it fails, showing them the grading code or the raised exception. This is done by activating the option *Show debug info*.



Tests

Setup code for all tests

```python
import numpy as np
```

+ Add a test

Name	Weight	Show debug info	Options
Weights	2	✓	✎ ✖
Gradient	1	✓	✎ ✖
Delta 1	1	☐	✎ ✖
Delta 2	1	☐	✎ ✖

Figure 3. Assignment's tests in the auto-grader configuration for Jupyter notebooks.

After saving the assignment, the *OK* configuration files are automatically generated, as well as some configuration files necessary to UNCode internals. These files are necessary to execute the submission using the configured tests, and with that, generate the corresponding feedback and grade, as explained in the next section.

3.2. Running Submissions and Generating Feedback

The second stage consisted of developing the environment where submissions are executed, generating the feedback and grade to be delivered to the student. When a student submits a proposal of a solution using UNCode, this submission is managed by the backend, which is in charge of managing all submissions and the corresponding submission queue. This backend sends the submission to another service called the Docker agent, which can be horizontally scaled as well. Then, this service is in charge of running the corresponding submission in a *Docker* container, which at the same time, runs each test in an additional docker container, acting as a sandbox to secure the student's submission. Currently, UNCode supports several grading environments for different kinds of assignments. Additionally, various submissions can be run in parallel to speed up the response time. In this context, the development of the present work was focused on creating a new grading environment for Jupyter notebooks. Therefore, and due to its architectural design, this new grading environment can be easily plugged in to UNCode. This is illustrated in Figure 4, where the different components are shown.

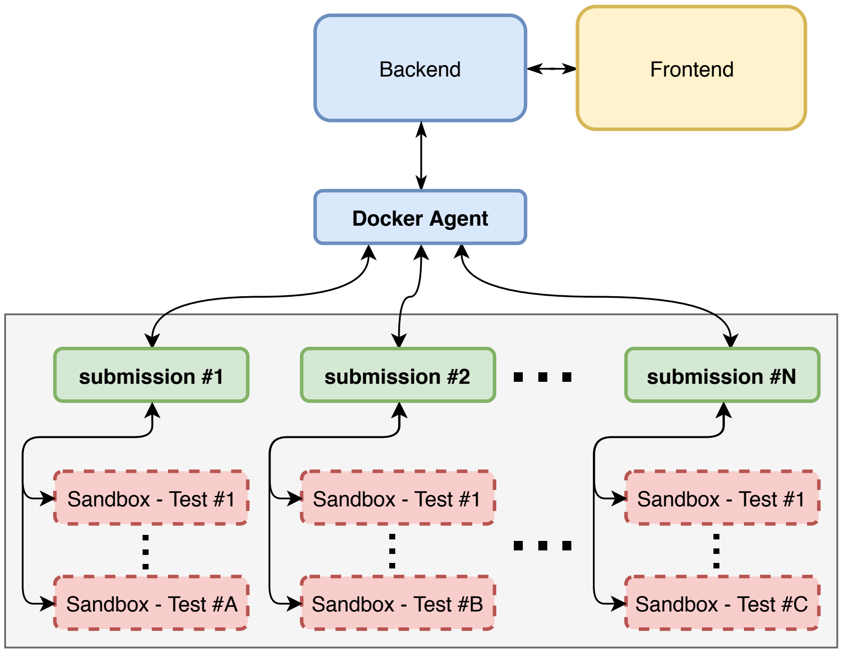


Figure 4. System architecture diagram corresponding to the grading environment on UNCode notebook. The backend is in charge of managing the queue of submissions, which sends the submission to the docker agent service and executes the submission in a sandboxed environment with Docker.

When this new grading environment or container is started, it already contains the configuration files of the assignment that were created automatically by the grader, as well as the submitted notebook. Additionally, this container already has installed all Python modules that might be necessary to run the student's code. The installed dependencies are commonly used Python modules in AI courses, for instance, Pandas, Scikit-learn, Keras, among others. On this basis, the student's notebook is executed inside the container to finally return the feedback and grade to the student in the front-end. To accomplish this, all phases involved within the grading environment are explained in more detail as follows.

3.2.1. Extracting Source Code

Initially, to be able to run the submission, source code is extracted from the submitted notebook; this is done with *nbconvert* (https://nbconvert.readthedocs.io, accessed on: 29

October 2021). This tool automatically allows us to convert *.ipynb* files to a Python script containing only the source code.

After the source code is in a Python script, this script is preprocessed. As Jupyter notebooks run over the *IPython* interactive environment [42], they support additional syntax constructs that are not valid under the Python execution environment. Thus, during the conversion process to a script, IPython code is generated. However, only Python code can be executed, and the IPython code must be removed. For instance, a Python module installation syntax needs to be removed from the script to correctly run it under the Python environment. Afterwards, an additional preprocess is accomplished; a *try-except expression* is added to enclose all lines of code. This is due to the fact that all the code is in a single script, then, some lines of code may throw exceptions and the grader will evaluate incorrectly the submission as not all tests could be executed. Here is shown an example of a caught exception:

```
try:
    a = 1 / 0
except:
    pass
```

This way of running notebook submissions has an advantage, students can create as many code cells as they want in the notebook, without the need of using labels or pragmas to indicate to the grader how to parse the code; this gives more flexibility to students when designing the solution. On the other hand, in case the submitted notebook is corrupted or indeed is not a notebook, the process finishes and an error message is shown to the student.

3.2.2. Running Grading Tests

Once the Python script is ready, each test can be executed using the *OK CLI*. For that, every test runs in an additional docker container that is launched from the grading container as shown in Figure 4, for instance, *Sandbox—Test #1*. That launched container for each test will act as a sandbox to provide security to the grading system. That way, it avoids the student's code to interact with the grading code. In addition, the container is created by passing memory and time limit parameters to determine when these types of errors occur during the execution of the tests.

3.2.3. Generating Summative and Formative Feedback

When the sandbox container finishes running the test, the generated standard output and error by OK are collected, then they are parsed to detect the test cases that have failed. In case a test case has failed, the executed code is obtained as well as either the output difference or the thrown runtime exception. This is done to show the student the additional debugging information that may lead them to eventually correct their proposal of solution. This process is done for all test cases. Next are described all possible results or feedback categories that the student may obtain from the generated feedback for each test and test case:

- *Accepted*: all test cases were successful in the given test.
- *Wrong answer*: when the student's code does not pass a test case and the obtained answer does not coincide with the expected output. The student might be able to see the executed grading code and the output difference in case this test is configured to show this additional debugging information.
- *Runtime error*: this result corresponds to a raised exception while running the student's code. In case the test is configured to show additional debugging information, the executed grading code and the raised exception are included within the feedback.
- *Grading runtime error*: this type of result is very similar to Runtime error, however, the raised exception occurred while the grading code was running; thus, the raised exception might be due to either there is indeed an error in the grading code that the instructor did not see while creating the task, or the student has not followed the assignment instructions, and the notebook solution does not have the functionality

to be tested, for instance, the student has changed the name of a function to be implemented. The executed grading code and the raised exception are sent back within the submission feedback.

- *Time limit exceeded*: the student's code took more time to finish than the allowed maximum time limit. Then, the whole test is marked with this result.
- *Memory limit exceeded*: a test obtains this error when the test code used more memory than the allowed maximum memory limit for the test.

At the same time the feedback is generated for each test, the grade is also calculated. For that, the number of accepted cases is counted, then the grade of a single test is obtained dividing the total passed test cases by the total of cases in that single test. After all tests have run, the total grade is calculated by the sum of each test weight times the test grade, then divided by the sum of all weights. After all this process, the generated feedback and the grade are sent back to the student. On UNCode notebook auto-grader, the summative feedback corresponds to the feedback category and the obtained grade, and the formative feedback corresponds to the executed code in the test and eventual thrown exceptions, all within the same feedback.

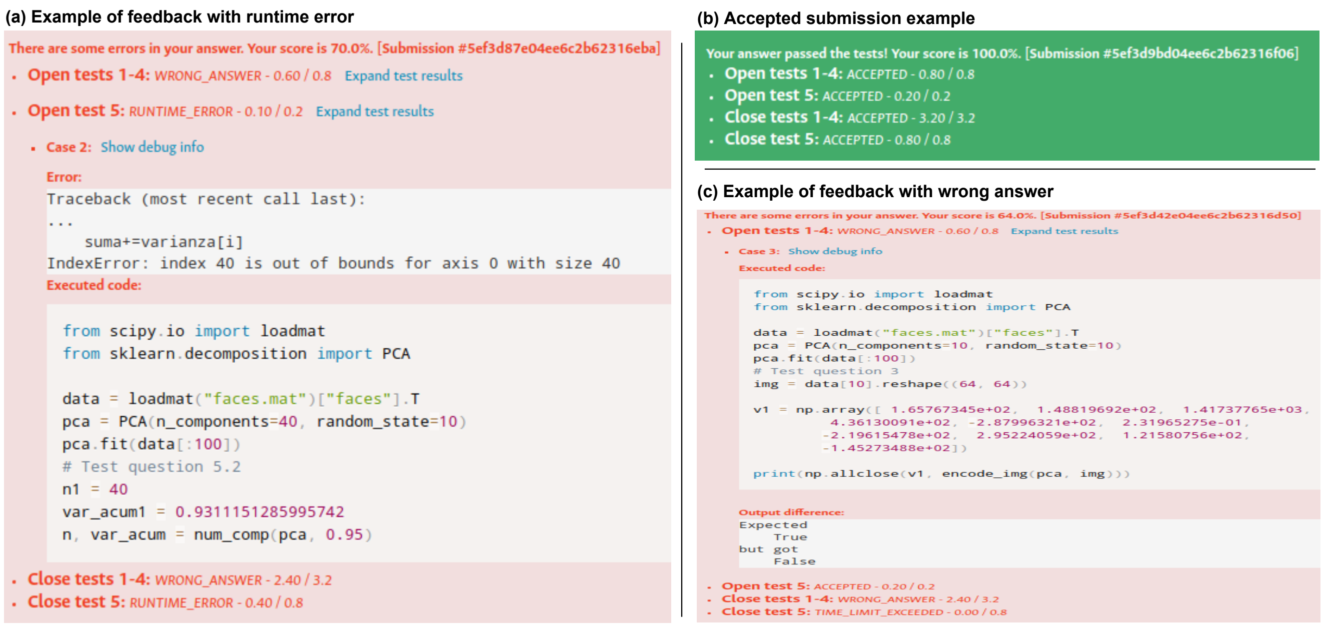
3.3. Bringing All Together

To sum up, initially the instructor creates the student version of the assignment, then, the automatic grader is configured adding all necessary *tests* and *test cases*, as well as additional datasets. After that, the student will either download the notebook template and solve it in their local computer, or open it and solve it in a cloud-based Jupyter notebook environment. When the student submits the solution notebook to UNCode notebook, they will instantly receive the feedback and partial grade for the given submission.

Figure 5 illustrates three different examples of feedback (summative and formative): subfigure (a) shows an example of a submission with grade 70.0% and some tests labeled as WRONG_ANSWER and RUNTIME_ERROR, corresponding to the summative feedback. One of the tests shows feedback with the details of the runtime error, where the grading code and raised exception are shown to the student (formative feedback). It is worth noting that other tests do not show additional details to the student, as the task was configured in that way. Subfigure (b) presents an example of an accepted submission, where all tests passed and the final grade is 100%. For this case, the only presented feedback is summative as all tests were correct and no further formative feedback was necessary. The subfigure (c) illustrates a submission with a grade of 64.0% and several feedback categories for each test in the submission (summative feedback). Here is also presented in more detail a test labeled as WRONG_ANSWER, where the grading code and output difference of the expected output and submission's output is shown (formative feedback).

3.4. Limitations

Due to the design decisions in the development stage of the proposed tool, UNCode Notebook auto-grader does not allow to use some IPython specific syntax. It is not possible to use magic commands (https://ipython.readthedocs.io/en/stable/interactive/magics.html, accessed on: 29 October 2021), the integrated shell to execute commands using the exclamation mark character, among other specific syntax introduced by Jupyter Notebooks (Python vs. IPython | https://ipython.readthedocs.io/en/stable/interactive/python-ipython-diff.html, accessed on: 29 October 2021). Moreover, the students are not allowed to install new modules, as the environment where the submissions run already have all the used and necessary modules to successfully run the student's notebooks. It is worth noting that in case these features are used in the notebook, the auto-grader will not fail; the tool automatically parses these special command lines before the Python code is executed.



(a) Example of feedback with runtime error

There are some errors in your answer. Your score is 70.0%. [Submission #5ef3d87e04ee6c2b62316eba]

- **Open tests 1-4:** WRONG_ANSWER - 0.60 / 0.8 Expand test results
- **Open test 5:** RUNTIME_ERROR - 0.10 / 0.2 Expand test results
  - **Case 2:** Show debug info

    Error:

    ```
    Traceback (most recent call last):
    ...
        suma+=varianza[i]
    IndexError: index 40 is out of bounds for axis 0 with size 40
    ```

    Executed code:

    ```
    from scipy.io import loadmat
    from sklearn.decomposition import PCA

    data = loadmat("faces.mat")["faces"].T
    pca = PCA(n_components=40, random_state=10)
    pca.fit(data[:100])
    # Test question 5.2
    n1 = 40
    var_acum1 = 0.9311151285995742
    n, var_acum = num_comp(pca, 0.95)
    ```
- **Close tests 1-4:** WRONG_ANSWER - 2.40 / 3.2
- **Close test 5:** RUNTIME_ERROR - 0.40 / 0.8

(b) Accepted submission example

Your answer passed the tests! Your score is 100.0%. [Submission #5ef3d9bd04ee6c2b62316f06]

- **Open tests 1-4:** ACCEPTED - 0.80 / 0.8
- **Open test 5:** ACCEPTED - 0.20 / 0.2
- **Close tests 1-4:** ACCEPTED - 3.20 / 3.2
- **Close test 5:** ACCEPTED - 0.80 / 0.8

(c) Example of feedback with wrong answer

There are some errors in your answer. Your score is 64.0%. [Submission #5ef3d42e04ee6c2b62316d50]

- **Open tests 1-4:** WRONG_ANSWER - 0.60 / 0.8 Expand test results
  - **Case 3:** Show debug info

    Executed code:

    ```
    from scipy.io import loadmat
    from sklearn.decomposition import PCA

    data = loadmat("faces.mat")["faces"].T
    pca = PCA(n_components=10, random_state=10)
    pca.fit(data[:100])
    # Test question 3
    img = data[10].reshape((64, 64))

    v1 = np.array([ 1.65767345e+02,  1.48819692e+02,  1.41737765e+03,
            4.36130091e+02, -2.87996321e+02,  2.31965275e-01,
           -2.19615478e+02,  2.95224059e+02,  1.21580756e+02,
           -1.45273488e+02])

    print(np.allclose(v1, encode_img(pca, img)))
    ```

    Output difference:

    ```
    Expected
        True
    but got
        False
    ```
- **Open test 5:** ACCEPTED - 0.20 / 0.2
- **Close tests 1-4:** WRONG_ANSWER - 2.40 / 3.2
- **Close test 5:** TIME_LIMIT_EXCEEDED - 0.00 / 0.8

Figure 5. Example of different feedback (summative and formative) the students might obtain after submitting their code: (**a**) feedback with runtime error, (**b**) accepted submission, and (**c**) wrong answer feedback for a given test case.

4. Materials and Methods

4.1. Setting

The developed UNCode automatic grader for Jupyter notebooks was employed to evaluate programming activities in two AI related courses at the Universidad Nacional de Colombia in the first semester of 2020: *Introduction to Intelligent Systems* and *Machine Learning*, supporting different kinds of in-class or extra-class programming activities. Each course proposed different programming activities to the students on UNCode throughout the courses. These programming activities can be divided in two categories: *assignment*, which is solved in a period of time of several hours or days and not necessarily solved during the class; *Quiz* is an examination activity to be solved during the class in short periods of time, generally two hours.

To be able to answer the research questions (RQ1 and RQ2), data was collected from UNCode to understand the students' interactions with the tool (i.e., students' submissions). In addition, a survey was conducted in both courses, which contained three multiple-choice and open-ended questions; this survey enabled us to recognize the students' perceptions about what they think about UNCode notebook auto-grader. Thus, we measured both the students' interaction with UNCode notebook auto-grader and their judgments about the tool. The collected data from both instruments were analyzed using descriptive statistics.

It is also important to mention that they are traditionally taught in person, although it was necessary to conduct both courses remotely due to COVID-19 pandemic and generalized lockdowns. Next, we explain in more detail the activities carried out per course, and their respective methodology.

4.1.1. Course: Introduction to Intelligent Systems

The course was developed based on two weekly sessions: lecture sessions, where the concepts are presented with the help of presentations and Jupyter notebooks; some sessions were merely practical, where students individually solved quizzes and assignments on UNCode. There were six programming activities supported on UNCode using Jupyter notebooks. These activities are described below in the chronological order that the students were able to start solving them:

- Assignment 1: the students were asked to implement various informed search algorithms, such as *Breadth First Search*.
- Assignment 2: the students trained some linear classification models using a benchmark dataset.
- Quiz 1: this quiz consisted on implementing from scratch a *Decision Tree model* and to train it.
- Quiz 2: the students implemented a small *feed-forward neural network*.
- Assignment 3: this was related to the topic *clustering*, where students developed a *K-Means* model.
- Assignment 4: students implemented the algorithm Principal Component Analysis (PCA) for dimensionality reduction for a series of images.

4.1.2. Course: Machine Learning

The methodology of this course is based on two weekly sessions, each session lasts around two hours. In lecture sessions, the concepts are presented with the help of presentations and Jupyter notebooks. Moreover, some sessions were merely practical, where students individually solved quizzes and assignments on UNCode. In that context, four different activities were supported on UNCode using Jupyter notebooks in Python, which are explained below in more detail, in the chronological order they were taken:

- Quiz 1: the students trained a logistic regression model using a public dataset.
- Assignment 1: the students implemented from scratch a *Kernel Ridge Regression model*. This homework was solved in three days.
- Quiz 2: this quiz consisted on implementing some methods related to the topic of *Kernel Logistic Regression*.
- Quiz 3: the students implemented a small *feed-forward neural network* from scratch.

4.2. Participants

The first course, *Introduction to Intelligent Systems*, is taught to undergraduate students in the Department of Systems and Industrial Engineering at the Universidad Nacional de Colombia. The course lasts 16 weeks. The goal of this course is to study the theory and the different methods and techniques used to create rational agents with a strong focus on machine learning. Some of the covered topics during the course include intelligent agents, informed and uninformed search algorithms, supervised learning, linear and nonlinear classification, neural networks and deep learning, and dimensionality reduction. A total of 41 students participated developing activities in UNCode.

The Machine Learning (ML) course is taught in the master’s degree in Systems and Computing Engineering at the Universidad Nacional de Colombia. It also lasts 16 weeks. The main goal of this course is to study the computational, mathematical, and statistical foundations of ML, which are essential for the theoretical analysis of existing learning algorithms, the development of new algorithms, and the well-founded application of ML to solve real-world problems. Some of the covered topics during the course included learning foundations, kernel methods, neural networks, deep learning, and probabilistic programming. A total of 35 students were enrolled in the course.

A total of 55 students from both courses participated in the survey to their perception of UNCode throughout the carried out activities. It is worth noting that the participation in the survey was at the will of each student, hence there are fewer students. On average, the participants were 24.4 years old at the time this survey was completed, with a standard deviation of 2.8 years. Most of the participants were male, representing 85.5% of the participants. Additionally, the students were enrolled in different programs: undergraduate degree in systems and computer engineering (76.3%), master’s degree in systems and computer engineering (9.1%), undergraduate degree in mechatronics engineering (3.6%), master’s degree in bioinformatics (1.8%), master’s degree in telecommunications (1.8%), undergraduate degree in physics (1.8%), undergraduate degree in industrial engineering (1.8%), master’s degree in applied mathematics (1.8%), and master’s degree in statistics

(1.8%). A total of 32 (out of 41) students of the *Introduction to Intelligent Systems* course participated in the survey, and the remainder 23 participants were enrolled in the *Machine Learning* course (out of 35 students).

4.3. Measurement Instruments

To answer the first research question (RQ1. How students interact with UNCode notebook auto-grader?) quantitative data were collected from UNCode for each one of the programming activities in the courses that were hosted on UNCode. The data collected included: number of participants in the courses and activities, number of submissions, as well as their respective results and the feedback category (e.g., accepted, runtime error, wrong answer). Thus, the tool itself was considered the first measurement instrument of the study.

Regarding to the second research question (RQ2. What is the students' perception of the tool as a support mechanism for their learning process?), a survey was conducted at the end of the semester. This survey aims to help the authors to understand better the students' thoughts and receive feedback about the tool, being this survey considered the second measurement instrument. To collect this data, the survey was created using the online tool *Google Forms* and all students from both courses were asked to complete it, although participation was at the will of each student. Moreover, an informed consent was given at the beginning of the survey to inform the students that the submitted data were treated anonymously, only used for research purposes and the answers did not affect the final course grade. This informed consent must be accepted or rejected by the students.

The survey was composed of three multiple-choice and open-ended questions, which were totally focused on the student's perception and usefulness of UNCode's Jupyter notebooks automatic grading. Each question was composed of two mandatory-to-answer parts: in the first part, the student selected one option from the Likert scale [43], which is composed of six levels of agreement/disagreement with respect to a given statement: totally disagree (1), disagree (2), somehow disagree (3), somehow agree (4), agree (5), totally agree (6). The second part was an open-ended question asking the reason they have selected the option of the already mentioned Likert scale. That way, we were able to know different levels of agreement about some features of Jupyter notebooks auto-grader on UNCode, as well as detailed perception and feedback from students. The statements included in the survey were:

1. *Statement 1*: I consider that the automatic grader offered by UNCode is a good mechanism to evaluate my performance in the course.
2. *Statement 2*: The automatic feedback provided by UNCode is useful to know how to correct errors in the solution to a given programming activity.
3. *Statement 3*: The UNCode's functionality that allows me to see the grading code of a test case is useful to debug my solution to the programming activity.

4.4. Data Analysis

The data collected from UNCode, corresponding to the first measurement instrument, was analyzed through descriptive statistics and the corresponding results are shown in Section 5.1; some bar charts and tables are presented to help visualization of results related to the students' interaction.

Moreover, the data collected from the survey was analyzed as follows. The quantitative data from the multiple-choice statements were analyzed via descriptive statistics, with the help of bar charts to visualize the number and percentage of students that selected a given level of agreement in the Likert scale from the survey. In addition, the conducted analysis on the open-ended questions was guided by the widely used framework Grounded Theory for qualitative data analysis, as described by Bryman [44], which is demonstrated to lead researchers to outstanding results. Following this framework, we encoded each of the answers given by the students, that is, detecting some key words that encode the whole answer. The generated codes work as indicators to detect concepts showing the impact of the UNCode notebook auto-grader on students' learning process. These concepts refer to labels given to the answers and are determined by grouping the codes by establishing common aspects and links between them, such us patterns of opinions. Nonetheless, it must be noted that an answer (student's opinion) may be identified not only for a single concept, but various concepts. The process of establishing and analyzing the concepts yields categories, which are at a higher level of abstraction than concepts, enclosing several concepts into one category with common characteristics, and therefore they globally represent students' perceptions.

The analysis on the open-ended questions was carried out in an iterative process and several times to correctly encode and categorize each answer to give more coherence to the interpretation of the answers. The analysis of the qualitative data was carried out by three of the main researchers in this work with the help of spreadsheets in which the perceptions given by the students were transcribed and ordered. It must be noted that the students' responses from both courses, in overall, denoted similar categories and opinions, and in consequence it was not pertinent to analyze and present the qualitative results separately for each course, but in a single sample of 55 students from both courses.

5. Results

5.1. Students' Interaction with UNCode (RQ1)

This section presents the results of the carried-out analysis over the data collected from UNCode related to the student's interaction with UNCode notebook auto-grader during the proposed activities, this corresponding to the first research question (RQ1).

5.1.1. Introduction to Intelligent Systems

Table 2 contains the number of students who participated in each activity carried out on UNCode, as well as the total students that succeeded in each activity. Additionally, the total number of submissions and submissions per student (i.e., the number of submissions divided by the number of students) are shown. The number of students that participated in each activity varied because not all students got to submit the solution, although they were trying to solve the activity.

Table 2. Number of students that participated, succeeded, and total submissions per activity and average submissions per student in each activity for the course Introduction to Intelligent Systems.

Activity	Students	Students Succeeded	Submissions	Submissions Per Student
Quiz 1	38	3	319	8.39
Quiz 2	37	20	216	5.84
Assignment 1	37	14	304	8.22
Assignment 2	38	29	272	7.16
Assignment 3	38	10	416	10.95
Assignment 4	38	35	317	8.34

A total of 38 students participated on each task on average (92.7%, out of 41). There were 1844 submissions in total, and on average, 307 submissions per activity. Moreover, each student made 7.5 submissions per activity on average. The activity with the most students that succeeded, with 35 (92.1%), was the *Assignment 4*, and the activity with the least students that succeeded, with only three students, i.e., 7.9%, was *Quiz 1*. This difference may have two reasons. The first one is the difficulty of each activity, the topic of each activity is different, thus, the *Quiz 1* was more difficult than the other activities. The second reason is the time given to solve each activity, as mentioned before, the solving-time of quizzes is shorter than assignments, then *Quiz 1* was difficult to be solved in the given time. This is not the case for *Assignment 4* or *Quiz 2*, where a good number of students succeeded in these activities.

Figure 6 illustrates the number of submissions for each feedback category described in Section 3.2.3. Each feedback category is shown in a specific color (accepted, runtime error, grading runtime error, time limit exceeded, memory limit exceeded, and wrong answer), and each bar represents the total submissions per category and activity. Note that green bars correspond to accepted submissions.

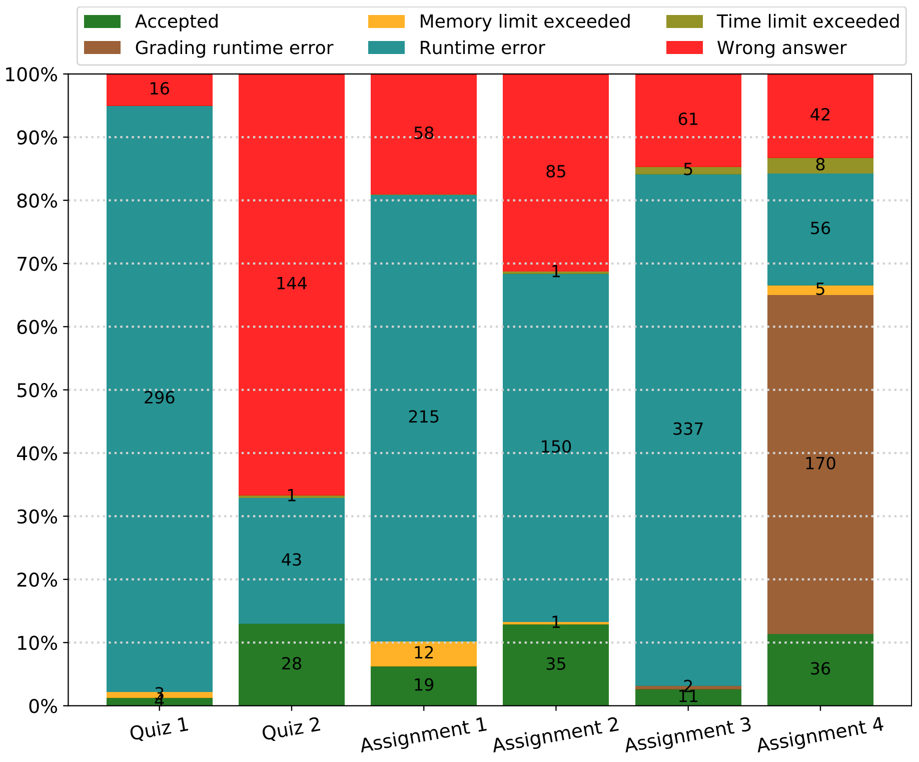


Figure 6. Number of submissions per activity and feedback category (i.e., accepted, runtime error, grading runtime error, time limit exceeded, memory limit exceeded, and wrong answer) for the course *Introduction to Intelligent Systems*.

In some cases, the number of accepted submissions is higher than the number of students that succeeded. This is because the students were free to submit several times the correct solution, though this was not common and it was only done by some few students. In overall, *memory limit exceeded* and *time limit exceeded* were not common among the activities, being 8 submissions with time limit exceeded (*Assignment 4*), and at most 12 submissions with memory limit exceeded for *Assignment 1*. This might be due to the fact that the activities were not assessing the performance, but the correctness of the solution, thus the memory and time limits were fair enough.

Moreover, it is worth noting that submissions that result in runtime errors in *Quiz 1*, *Assignment 1*, *Assignment 2*, and *Assignment 3* correspond to more than the 50% of submissions for these activities, corresponding to 92.8%, 70.7%, 55.1%, and 81.0%, respectively, from the total number of submissions. This is due to the way the activities are designed and how students test their proposal of solution. As the activities are divided into several programming exercises, when the students think they have solved a specific exercise, they make a submission; however, as the other exercises are not solved yet, it will raise a runtime error as the code is still executed. For that reason, the number of wrong answer submissions is not large for those activities. It is also worth to mention, even though a submission in general is categorized as runtime error, the student also may see some test cases with other feedback categories. Moreover, something quite noticeable is the large amount of grading runtime errors for *Assignment 4*, corresponding to 53.6% of the total submissions in this activity. The explanation of this is quite similar to runtime errors in the other activities, where students make several submissions when they solve a specific exercise, although, this is more related to the way the grading code was designed by the instructor; this is, the code that checks if the student's code returns the desired result or not. In this case, many implemented solutions did not return the expected data type in the grader, and therefore, exceptions were thrown by the grading code. This is also a guide for students to return the correct value and data type. With respect to *Quiz 2*, it has the particularity that around 66.7% of submissions correspond to wrong answers. In this case, when students did not solve an activity's exercise or the value was wrong, the grading code correctly managed this, thus, in most of submissions, the students obtained wrong answers.

5.1.2. Machine Learning

Table 3 shows the number of students that participated in each activity on UNCode notebook auto-grader, this is, the students that did at least one submission for the given activity. Moreover, the number of students that succeeded, the total submissions per activity, and the average of submissions per student (i.e., the number of submissions divided by the number of students) for each activity are also presented.

Table 3. Number of students that participated, succeeded, and the number of submissions per activity and average submissions per student in each activity for the course *Machine Learning*.

Activity	**Students**	**Students Succeeded**	**Submissions**	**Submissions Per Student**
Assignment 1	30	29	174	5.80
Quiz 1	34	23	275	8.08
Quiz 2	31	4	154	4.97
Quiz 3	31	14	229	7.39

In general, 32 students participated in each task on average, corresponding to the 91.4% of the total students that participated. There were 832 submissions in total, and on average around 208 submissions per activity. In that way, each student carried out 5.9 submissions per activity. The activity with the most students that succeeded was *Assignment 1*, with 29 (96.7%) different students, and the activity with the least students that succeeded was *Quiz 2*, corresponding to only 4 (12.9%) students out of 31. The explanation of these results is very similar as in the other course, it is a combination of complexity and the time allowed to solve the activity. For instance, *Quiz 2* is more comparable to the other quizzes, which had a similar time limit, while the other quizzes were solved by more students, *Quiz 2* was more complex, and the given time was too short to be correctly solved by more students.

Figure 7 displays a stacked bar chart with the number of submissions done for each activity; each bar represents the number of submissions for each activity in the given feedback category. Firstly, there were no time limit exceeded submissions and also the number of memory limit exceeded submissions was small. This is due to the fact that the activities had a large time limit, as well as the memory limit. Furthermore, the activities were not computationally extensive and most of the time, the solutions were optimal. Something else to note is that the number of accepted submissions is greater than the number of students that succeeded. As previously mentioned, students are allowed to send several times their proposal of solution, even if they were already correct, though, in this case it is more common. For instance, 23 students succeeded in *Quiz 1*, but there were 13 accepted submissions more; in this case, students probably were trying different ways to solve the problem.

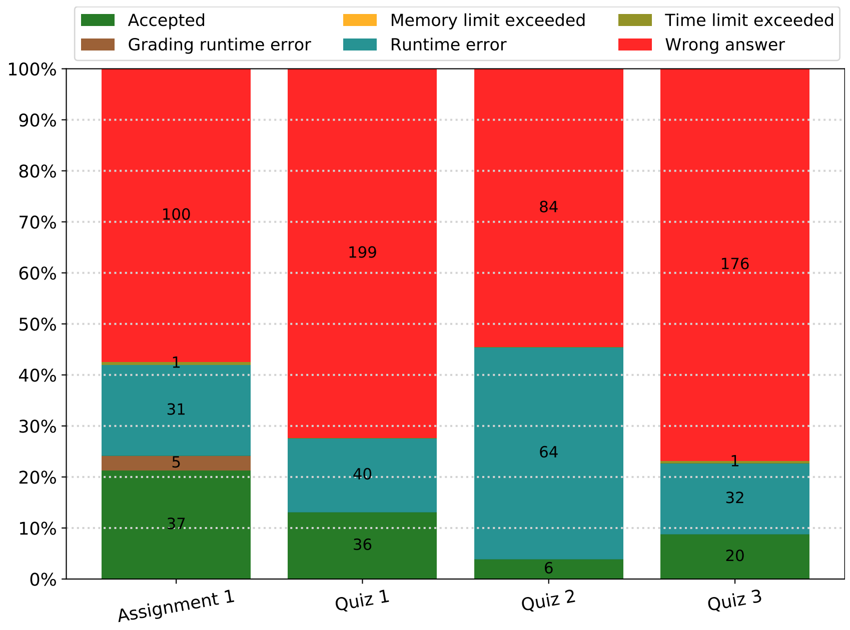


Figure 7. Number of submissions per activity and each feedback category (accepted, runtime error, grading runtime error, time limit exceeded, memory limit exceeded, and wrong answer) for the *Machine Learning* course.

Another particularity seen in Figure 7 is the small number of grading runtime errors, being only 5 for *Assignment 1*. In this case, the grading code managed invalid values and incorrect types of variables returned by the students' code. Thus, these errors are due to the students, as they did not follow the instructions and the grader could not find the graded functions in the students' notebook. Moreover, in all the activities the amount of submissions cataloged as wrong answer correspond to more than half of the submissions; this is around 57.5% for *Assignment 1*, 72.4% for *Quiz 1*, 54.5% for *Quiz 2*, and 76.8% for *Quiz 3* of the total submissions per activity. The reason for this is that, as mentioned before, the activities were split into several exercises, and the students tended to do a submission every time they solved a single exercise, thus, they obtained in overall a wrong answer as the other exercises were not solved yet. Furthermore, the grading code was designed to return a wrong answer for these cases. Nonetheless, runtime errors did not represent a large number of submissions. It is worth mentioning that it is much smaller than the other course. This could be explained by the students' previous experience with Jupyter notebooks and automatic graders at the moment of solving the activities.

5.2. Students' Perception and Feedback (RQ2)

Figure 8 illustrates in a stacked bar chart, the answers of the students for the levels of agreement/disagreement in the Likert scale for the three statements in the survey previously described. Among the 55 respondents, the majority of students have indicated some level of agreement (4—Somehow agree; 5—Agree; or 6—Totally agree) in all statements, where the 94.5% of respondents agree that the automatic grader of notebooks on UNCode was a good mechanism to evaluate their performance (statement 1), 81.8% agree that the automatic feedback was useful for debugging (statement 2), and 96.4% agree that showing the grading code in the feedback is useful to debug the solution (statement 3). Nevertheless, some students also expressed some level of disagreement, corresponding to 5.5%, 18.2%, 3.6%, respectively, for the three statements in the survey.

Next, we present the results of the analysis of the answers of the students to the three open-ended questions of the survey.

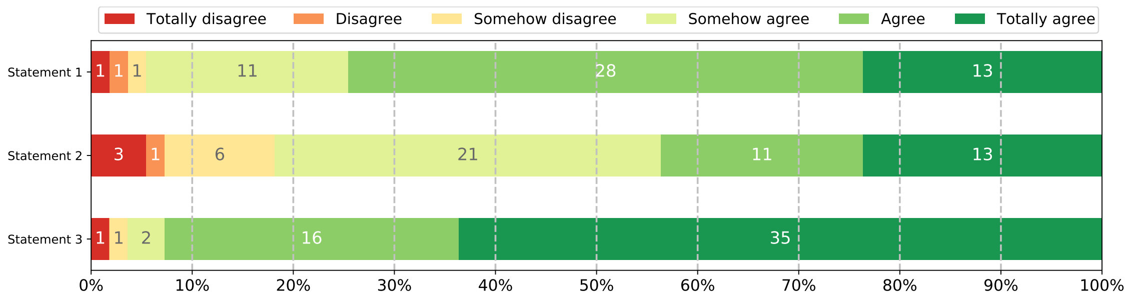


Figure 8. Levels of agreement/disagreement corresponding to the answers of students in the Likert scale for the three statements of the survey.

5.2.1. *Statement 1*: I Consider That the Automatic Grader Offered by UNCode Is a Good Mechanism to Evaluate My Performance in the Course

Following the process described in Section 4.4 to categorize the answers to open-ended questions, seven different categories were identified in the students' perceptions. It is also worthy to note that not all perceptions were exclusively classified within only one category, as they might be classified within another category, being this applicable to all the questions. Table 4 summarizes all the categories, the description, a sample quote of an opinion expressed by a student, and the number of opinions within this category extracted from the students' answers.

Table 4. Categories defined from the students' answers for statement 1.

Category	Description	Sample Quote	Number of Opinions within This Category
Precise and impartial testing	In some of the answers the students expressed that the testing system is precise and impartial, as the expected solutions are exact; thus, the activities are objectively evaluated. Additionally, some answers were focused on the option to make several submissions, and it makes the testing system fair, as they expressed.	*"It is impartial and if one correctly solves the activity, the deserved grade is given"*.	10
Immediate and helpful feedback	Some opinions of the students were focused on the feedback, as it was immediate and helpful to solve the problem; this as a consequence of showing additional information about the tests.	*"As it returns immediate feedback and gives additional information about the test cases, it allows me to solve the activities and successfully fix errors"*.	9
Useful as an evaluation mechanism	The auto-grader indeed was useful as a mechanism to evaluate their performance in the course.	*"I believe that the tool properly works as an evaluation mechanism"*.	6
Test knowledge	They opined that the automatic grader allows them to test their knowledge and understanding regarding the in-class taught concepts, as they were able to practice the theory.	*"It evaluates, in a practical way, whether the concepts were correctly understood or not"*.	5
Immediate grade	Due to the fact that a grade is given every time a submission is done, the students expressed that UNCode's auto-grader is a good mechanism to evaluate their performance, as they constantly monitored their performance throughout the course with the given grade.	*"It allows me to know immediately my grade"*.	4
Learn from errors	The participants mentioned that the auto-grader allows them to learn from errors as they see errors in their proposed solutions thanks to the feedback and they can correct their solutions to get to solve the activity.	*"It allows us to quickly understand where the code is failing and gives us a general idea of how to solve it"*.	8
Automatic grading is not enough	The students mentioned that even though the automatic grade was good, it is not enough to evaluate the whole performance throughout the course. Their opinions fix attention on additional ways to evaluate, such as manual grading, or other kinds of metrics that can be used to evaluate performance, i.e., proximity to the expected answer; they think it might be more helpful or better.	*"The evaluation of performance is reduced to a single number, but it does not take into account the carried out process and the put effort throughout the course"*.	12

To summarize, six out of the seven categories were positive and a good number of students are in symphony with this statement. However, some respondents wrote that an automatic grader is not enough to evaluate their performance, the reason for this might be due to the dynamics and the course methodology, as the unique way to evaluate these activities was using the UNCode's automatic grader, and therefore their only grade was given by UNCode. However, the students expressed that additional evaluation options and metrics to evaluate performance, such as proximity to the expected solution, effort and the outlined solution, among others, may be employed in the course's methodology. That is why not many students totally agree with this statement, and more students that agree or somehow agree, as shown in Figure 8.

5.2.2. *Statement 2*: The Automatic Feedback Provided by UNCode Is Useful to Know How to Correct Errors in the Solution to a Given Programming Activity

In the qualitative analysis of the answers to this statement asking why they agree/disagree with this statement, we have detected seven main categories to classify the perceptions, which are presented in Table 5.

From the analysis of this statement and the identified categories, we see that indeed it was useful to correct errors in students' proposal of solution, with five out of seven positive categories. Nevertheless, the opportunities of improvement were identified in two categories, that is, *Unclear test cases* and *Feedback could be improved*. Even though the *Unclear test cases* category is about grading code mainly, this is included in this paper as the way it is designed highly affects the perception of the usefulness of the automatic feedback. Furthermore, various improvements to the feedback. Therefore, there is more dissent about this statement, this in concordance with Figure 8, where 10 respondents in some way disagree, and also 21 of them somehow agree, which is reflected in the given answers.

5.2.3. *Statement 3*: The UNCode's Functionality That Allows to See the Grading Code of a Test Case Is Useful to Debug My Solution to the Programming Activity

We detected five main categories to group the different perceptions related to the usefulness of showing the grading code to debug the proposed solution, using the same analysis as in the previous two statements. Although some categories are similar as in *Statement 2*, we separate both statements as most of the answers are different and this statement only focuses on the usefulness of showing the grading code, rather than the feedback in general. These categories are presented in Table 6.

Table 5. Categories defined from the students' answers for statement 2.

Category	**Description**	**Sample Quote**	**Number of Opinions within This Category**
Useful to identify errors	The students agreed that it was helpful to spot errors in their code, which they later fixed.	*"The automatic feedback enabled me to know where the errors are"*.	13
Useful to solve errors	In addition to being helpful to identify errors, as the previous category, the automatic feedback was also useful to solve the errors in their proposal of solution. It should be noted that some answers were classified within this and the previous category, although it was not the case for all of them as they only talked about how helpful this was to solve the errors.	*" The tests work to verify and fix the code immediately"*.	6
Shows grading code	The students were able to see the grading code of some tests, which they considered helpful to test their code.	*"When something goes wrong, it shows the grading code..."*	6
Useful to identify not considered test cases and compare answers	It was useful to understand and see test cases that they did not consider before, also, as they see the expected output, they can compare the answers and detect where they are failing with respect to the expected answer.	*"It allows to compare and see the difference of the answers"*.	6
Good guide to solve the problem	In overall, the automatic feedback guided them towards the solution of the problem as they were able to identify some hints on the feedback, as well as to remember the theoretical concepts.	*"The feedback was very helpful, as it guides the person to remember concepts that may help to solve the problem"*.	4
Unclear test cases	The test cases were not clear and complex to understand for the students in some cases. Thus, the usefulness of the feedback is very contingent upon the way the grading code is designed.	*"As the test cases seem to be complex, then it is not possible to retrieve too much information from them..."*	8
Feedback could be improved	The answers associated with this category express and suggest that the feedback needs some improvements, such as adding more details when the submission fails, custom feedback set by the instructor that shows some additional hints, manual grading, among other improvements. 18 answers were cataloged within this category. It is also worth noting that while UNCode was employed to evaluate the programming activities, some bugs were detected and fixed, as well as some improvements in the feedback were made.	*"It could give some hints as right now the only presented information is via the grading code"*.	18

Table 6. Categories defined from the students' answers for statement 3.

Category	Description	Sample Quote	Number of Opinions within This Category
Good guide to solve the problem	The answers categorized within this category mention that the grading code as a good guide, which they can use to correctly solve the problem.	*"…this is the functionality that helps the most with the correct development of the activities"*.	19
Identify errors	The students opined that the grading code helped them to identify where they had errors.	*"It allows me to identify where and which the errors are"*.	22
Identify not considered test cases	The grading code helped these respondents to see edge cases, cases they did not consider before, as well as to understand how to implement the solution and return the expected answer.	*"…this is very useful, as it allows to evince several factors that could be omitted…"*	15
Run locally test cases	The respondents could debug their code, as they were able to copy and run the grading code in their development environment.	*"I could replicate the grading code in my local environment"*.	4
Test cases could be improved	The test cases could be improved, as for some few of them, it was more confusing than helpful because the grading code was not very clear sometimes and they had to spend more time trying to understand the code. Thus, the way the grading code is designed will directly affect the way students perceive the usefulness of the grading code, and in the same way, how they solve the activity.	*"It was useful, but it may take a while trying to understand the grading code, especially in activities with short periods of time"*.	7

As per Figure 8, where 35 respondents totally agree, it is noted that a vast majority of students consent to agree with this statement, as they see it was helpful to debug the proposal of solution. Moreover, only one out of five categories was identified to group perceptions related to improvements that can be done in test cases, even though this is more related to the instructor and how the activity is designed, it is important to note that the usefulness is not only dependent upon UNCode.

6. Discussion

To validate the usefulness of the UNCode notebook auto-grader in an academic context, we designed a study in two AI-related courses at the Universidad Nacional de Colombia. This experience allowed us to answer two research questions: (RQ1) How do students interact with the UNCode notebook auto-grader?; (RQ2) What is the students' perception of the tool as a support mechanism for their learning process?

Regarding the first research question (RQ1), our findings indicate that there was a great number of submissions in both courses. This is due to the possibility that the students could perform multiple attempts to solve the activities obtaining immediate feedback from the tool; it was advantageous because it allowed them to obtain formative feedback multiple times, which helps them to get close to the solution for the given activities. In the case of obtaining feedback informing an error in the proposed solution, students could identify aspects that need correction, with multiple opportunities to submit new versions of their solution, which is an important benefit. Contrarily, without the UNCode notebook auto-grader, the students would have a limited number of opportunities to submit their solutions for evaluation and the teaching staff would not be able to cope with this amount of submissions to be graded manually, without the support of the automatic tool. In fact, Ala-Mutka [14] identified that allowing multiple attempts if the student is not satisfied with the results of the programming assignment can be considered as formative assessment. In this case, this might help students by providing feedback on their work and let them improve it. In this way, by using the UNCode Notebook auto-grader, it is possible to effectively provide immediate and formative feedback to the students for a great number of submissions. In addition, the results also indicate the category of feedback assigned to a submission highly depends on how the instructor designs the grading code and the assignment itself, since possible runtime errors can be managed in the grading code and better error messages can be shown; this could be achieved by the instructors with more experience with the tool. Moreover, the succeeding rate of a given activity is contingent upon the difficulty and time restrictions, which influences the kind of feedback the students obtain.

With respect to the students' perception of UNCode notebook auto-grader, corresponding to the second research question (RQ2), the general opinion of the students regarding the tool is positive. Most of the students agreed that UNCode notebook auto-grader was a good mechanism to evaluate their performance, as well as agreed with the provided formative feedback was quite helpful for solving the programming assignments, and in general, their learning process. Many students mentioned that the feedback helped them to debug and fix their code, highlighting some features, such as: immediate formative and summative feedback, precise and impartial testing, possibility to obtain the grading code, multiple opportunities to submit, among others. In addition, they opined that the tool was useful to identify and solve errors, to guide them towards the solution, and to help them to detect unconsidered test cases, all of this being quite valuable to the practitioners. Although some limitations and a few aspects to improve were detected from negative opinions, such as the need for more detailed feedback, custom feedback and manual grading; it is important to mention that we already have addressed some of these suggestions to improve the system and response better to the learning process of the students, for instance: first, it is now possible for instructors to add custom feedback to the test cases, second, a new feature was developed to add manual reviews to students' submissions, among other improvements. In this sense, UNCode notebook auto-grader is not only useful as an automatic grading tool that provides summative feedback, but it is also a tool that offers formative feedback through different mechanisms that support the students during their learning process, thus facilitating them to find and correct errors.

Regarding the technical challenges from other automatic grading tools for Jupyter Notebooks identified in Section 2.1, the proposed tool provides a user interface for instructors to set up the grader and grading code. Unlike other tools (*nbgrader* [18], *Web-CAT* [22], *Vocareum*, among others), it does not depend on the JupyterHub Server, which reduces the necessary computational resources and costs to deploy. According to the task configuration selected by the instructor, the tests are generated automatically within the same platform, this in contrast with *Otter-Grader* and *OK*. Then, students are able to either download the notebook or open it using a development environment, like Colab, to start solving the programming assignment proposed in the notebook. When a student submits an attempt of solution on the online UNCode notebook auto-grader, it is executed in a secure sandboxed environment, which is either not provided by default or it is not used in other systems like nbgrader, OK, Web-CAT, this being important to make sure the students do not interact with the grading code. Afterwards, the feedback and grade are sent back immediately to the student and within the same platform. Therefore, a student can see her or their summative feedback with additional formative information, e.g., the grading code for some test cases. To highlight from the generated summative feedback, this tool addresses some detected lacks in the other generated feedback on tools like OK, as expressed by Sharp et al. [15], *CoCalc*, and *Coursera*. This was addressed by labeling the feedback with defined categories, detecting errors in the grading code, showing the raised exceptions, detecting time and memory limit exceeded errors, among others.

Concerning the summative evaluation, as Gordillo [29] points out, the participants in this experience emphasized that the automatic evaluation of their performance provided them with inputs to increase their practical problem-solving skills. Additionally, they pointed out that automatic grading has characteristics like being accurate, impartial, immediate, allows evaluating the knowledge and level of understanding of the theoretical concepts covered in the courses, and facilitates the permanent monitoring of performance in the subjects. These characteristics indicate specific advantages of including automatic summative assessment tools not only in computer programming learning environments, but also artificial intelligence courses like in this study.

From the analysis of students' interaction and perception, we see that a formative feedback is essential for the students to support them during their learning process. The participants in this experience pointed out as advantages of the formative evaluation the possibility of identifying errors in the code, making corrections quickly, solving problems correctly, and testing their programs by adding test cases not initially considered. As for the opinions in which the participants highlighted the need to add more information offered by the tool when programs fail, these confirm the need to continue increasing and complementing formative feedback in computer programming learning environments, as pointed out by Keuning et al. [27]. However, it is also critical, and recommended to the practitioners, to correctly design the test cases and give precise instructions, as this is a key point where students may perceive the grading system as either useful or not. All of this highlights why adopting an auto-grader for Jupyter Notebooks is useful and eases the grading process in computer science courses, either developing a new system or using one of the current grading systems. The Jupyter automatic notebook grading tool presented in this paper provides students with immediate summative and formative feedback, more specifically following the recommendations given by Ullah et al. [39], as the students are able to instantaneously see the location of their errors, as well as to run the grading code for a better understanding of their errors, which gives students a scaffolding to know how to proceed to correct their errors, which must be taken into account by practitioners and academicians.

7. Conclusions

We presented *UNCode notebook auto-grader*, where students can obtain not only summative feedback related to programming assignments but also detailed formative feedback in an instantaneous way. We added support to grade automatically Jupyter Notebooks on top of UNCode, the already working auto-grader at the Universidad Nacional de Colombia. To validate this new automatic grading tool, we also reported the experience of using the UNCode Jupyter notebook auto-grader in two AI courses: Introduction to Intelligent Systems and Machine Learning. The results of the study were divided into two parts: students' interaction with UNCode, and students' perception and feedback.

The possibility of obtaining summative and formative feedback automatically from an online tool is an important advantage for students when developing their solutions to challenging computer programming tasks in Jupyter Notebooks for AI courses. This was shown in both quantitative and qualitative data analysis conducted in this work. The large number of submissions made by the students, and the feedback obtained automatically guided them towards solving the proposed activities. Otherwise, without the support of an auto-grading tool, students might have a limited number of opportunities for the evaluation of their solutions, and the instructors would not be able to cope with a large number of submissions to be manually graded. Not to mention that, in such case, feedback could be communicated several days/weeks after the submission. Moreover, students' perceptions indicated that the proposed tool was adopted as a good mechanism to evaluate their performance; the feedback provided was useful not only to identify errors in the proposed solutions, but also to correct them, and the functionality developed to improve debugging information (by showing the grading code of some selected test cases to students) was also useful in the problem-solving process.

This work makes two main contributions to the area of computer science and engineering education: first, we introduce a publicly available Jupyter Notebooks auto-grading tool, which provides instructors with an easy-to-use automatic grader that offers summative and formative feedback to students instantaneously; and second, we provide empirical evidence on the benefits of using the proposed tool in an academic through the reported experience on the use of the tool in two AI courses, where the students expressed how helpful UNCode notebook auto-grader was for their learning process.

It should be noted that since this work presents a case study in two AI courses, the generalization of the results could be limited. Firstly, it is worth noting that the test had some particularities: not all the students in both courses participated in the survey, thus the number of samples was smaller. Secondly, mainly young male with a background in computation participated in the test. Further experimental design studies are recommended to provide solid evidence on the impact of the use of this novel tool on variables, such as academic performance and student motivation. Additionally, a quasi-experimental study can be carried out to compare the students outcomes of an experimental group using UNCode notebook with the results of a control group, and supplementary this might determine other information like whether the students learn more or not and how fast they learn, among others. Furthermore, future studies could focus on evaluating which type of feedback helped students debug their work, determining when they fixed an error (how many submissions they made until they fixed their code), and detecting the occasions the students were able to fix a specific error identified in the feedback. For future work, we plan to explore some of these possibilities. In addition, we will develop new functionalities to the tool to further improve the feedback provided to students, for example, by including some specific hints, pre-configured by the instructors, depending on the specific test case that is failing. Another way to send the notebooks directly from the development environment may also be developed to facilitate the students' workflow.

Author Contributions: Conceptualization, C.D.G.-C., F.R.-C. and F.A.G.; methodology, C.D.G.-C., F.R.-C., J.J.R.-E. and F.A.G.; software, C.D.G.-C.; validation, C.D.G.-C., F.R.-C. and F.A.G.; formal analysis, C.D.G.-C., F.R.-C. and J.J.R.-E.; investigation, C.D.G.-C., F.R.-C., J.J.R.-E. and F.A.G.; resources, F.R.-C. and F.A.G.; data curation, C.D.G.-C.; writing—original draft preparation, C.D.G.-C. and F.R.-C.; writing—review and editing, J.J.R.-E. and F.A.G.; visualization, C.D.G.-C.; supervision, F.R.-C.; project administration, F.R.-C.; funding acquisition, F.R.-C., J.J.R.-E. and F.A.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Universidad Nacional de Colombia grant number HERMES 48098.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to data privacy of the tool and students.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RQ	Research Question
ML	Machine Learning
AI	Artificial Intelligence
ICPC	International Collegiate Programming Contest
FDA	Flexible Dynamic Analyzer
GUI	Graphical User Interface
CLI	Command Line Interface
API	Application Programming Interface
LMS	Learning Management System
MOOC	Massive Open Online Course
HDL	Hardware Description Language
AGPL	Affero General Public License
Colab	Google Colaboratory
PCA	Principal Component Analysis

References

1. Barba, L.A.; Barker, L.J.; Blank, D.S.; Brown, J.; Downey, A.B.; George, T.; Heagy, L.J.; Mandli, K.T.; Moore, J.K.; Lippert, D.; et al. *Teaching and Learning with Jupyter*; Jupyter for Education—GitHub: San Francisco, CA, USA, 2019; p. 77.
2. Perkel, J.M. Why Jupyter is data scientists' computational notebook of choice. *Nature* **2018**, *563*, 145–146. [CrossRef] [PubMed]
3. Bisong, E. *Building Machine Learning and Deep Learning Models on Google Cloud Platform*; Apress: Berkeley, CA, USA, 2019; p. 709. [CrossRef]
4. Cabrera Granado, E.; García Díaz, E. *Guide To Jupyter Notebooks for Educational Purposes*; Universidad Complutense de Madrid: Madrid, Spain, 2018.

5. Raju, A.B. IPython Notebook for Teaching and Learning. In Proceedings of the International Conference on Transformations in Engineering Education, Hubli India, 16–18 January 2014; Natarajan, R., Ed.; ICTIEE 2014; Springer: New Delhi, India, 2015; p. 611._91. [CrossRef]
6. Brunner, R.J.; Kim, E.J. Teaching Data Science. *Procedia Comput. Sci.* **2016**, *80*, 1947–1956. [CrossRef]
7. Richardson, M.L.; Amini, B. Scientific Notebook Software: Applications for Academic Radiology. *Curr. Probl. Diagn. Radiol.* **2018**, *47*, 368–377. [CrossRef] [PubMed]
8. Reades, J. Teaching on Jupyter. *Region* **2020**, *7*, 21–34. [CrossRef]
9. Niederau, J.; Wellmann, F.; Maersch, J.; Urai, J. Jupyter Notebooks as tools for interactive learning of Concepts in Structural Geology and efficient grading of exercises. Geophysical Research Abstracts. In Proceedings of the EGU General Assembly Conference Abstracts, Vienna, Austria, 23–28 April 2017; Volume 19, p. 17191. [CrossRef]
10. Golman, B. Transient kinetic analysis of multipath reactions: An educational module using the IPython software package. *Educ. Chem. Eng.* **2016**, *15*, 1–18. [CrossRef]
11. Cardoso, A.; Leitão, J.; Teixeira, C. Using the Jupyter Notebook as a Tool to Support the Teaching and Learning Processes in Engineering Courses. In *The Challenges of the Digital Transformation in Education*; Auer, M.E., Tsiatsos, T., Eds.; Springer International Publishing: Cham, Switzerland, 2019; Volume 917, pp. 227–236._22. [CrossRef]
12. Alhazbi, S. Using e-journaling to improve self-regulated learning in introductory computer programming course. In Proceedings of the 2014 IEEE Global Engineering Education Conference (EDUCON), Istanbul, Turkey, 3–5 April 2014; IEEE Computer Society: Istanbul, Turkey, 2014; pp. 352–356. [CrossRef]
13. Cheang, B.; Kurnia, A.; Lim, A.; Oon, W.C. On automated grading of programming assignments in an academic institution. *Comput. Educ.* **2003**, *41*, 121–131. [CrossRef]
14. Ala-Mutka, K.M. A Survey of Automated Assessment Approaches for Programming Assignments. *Comput. Sci. Educ.* **2005**, *15*, 83–102. [CrossRef]
15. Sharp, C.; Van Assema, J.; Yu, B.; Zidane, K.; Malan, D.J. An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, Bali, Indonesia, 17–19 June 2020; ITiCSE '20; Association for Computing Machinery: Trondheim, Norway, 2020; pp. 487–492. [CrossRef]
16. Fonte, D.; Da Cruz, D.; Gançarski, A.L.; Henriques, P.R. A Flexible Dynamic System for Automatic Grading of Programming Exercises. In Proceedings of the 2nd Symposium on Languages, Applications and Technologies, Porto, Portugal, 20–21 June 2013; Leal, J.P., Rocha, R., Simões, A., Eds.; OpenAccess Series in Informatics (OASIcs); Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2013; Volume 29, pp. 129–144. [CrossRef]
17. Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli, Finland, 28–31 October 2010; Koli Calling '10; Association for Computing Machinery: Koli, Finland, 2010; pp. 86–93. [CrossRef]
18. Jupyter, P.; Blank, D.; Bourgin, D.; Brown, A.; Bussonnier, M.; Frederic, J.; Granger, B.; Griffiths, T.; Hamrick, J.; Kelley, K.; et al. nbgrader: A Tool for Creating and Grading Assignments in the Jupyter Notebook. *J. Open Source Educ.* **2019**, *2*, 32. [CrossRef]
19. Land, D. Automatic Grading in Engineering Classes. In Proceedings of the 10th International Conference on Physics Teaching in Engineering Education PTEE 2019, Delft, The Netherlands, 23–24 May 2019.
20. Hamrick, J.B. Creating and Grading IPython/Jupyter Notebook Assignments with NbGrader. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, TN, USA, 2–5 March 2016; SIGCSE '16; Association for Computing Machinery (ACM): Memphis, TN, USA, 2016; p. 242. [CrossRef]
21. Sridhara, S.; Hou, B.; Denero, J.; Lu, J. Fuzz Testing Projects in Massive Courses. In Proceedings of the Third (2016) ACM Conference on Learning @ Scale, Edinburgh, Scotland, UK, 25–26 April 2016; L@S '16; Association for Computing Machinery: Edinburgh, Scotland, UK, 2016; pp. 361–367. [CrossRef]
22. Manzoor, H.; Naik, A.; Shaffer, C.A.; North, C.; Edwards, S.H. Auto-Grading Jupyter Notebooks. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA, 11–14 March 2020; SIGCSE '20; Association for Computing Machinery: Portland, OR, USA, 2020; pp. 1139–1144. [CrossRef]
23. Skalka, J.; Drlik, M.; Benko, L.; Kapusta, J.; Del Pino, J.C.R.; Smyrnova-Trybulska, E.; Stolinska, A.; Svec, P.; Turcinek, P. Conceptual framework for programming skills development based on microlearning and automated source code evaluation in virtual learning environment. *Sustainability* **2021**, *13*, 3293. [CrossRef]
24. Narciss, S. Feedback strategies for interactive learning tasks. In *Handbook of Research on Educational Communications and Technology*; Routledge: Oxfordshire, UK, 2007; pp. 125–144. [CrossRef]
25. Restrepo-Calle, F.; Ramírez-Echeverry, J.J.; Gonzalez, F.A. UNCode: Interactive System for Learning and Automatic Evaluation of Computer Programming Skills. In Proceedings of the 10th International Conference on Education and New Learning Technologies, Palma, Spain, 2–4 July 2018; EDULEARN18 Proceedings; IATED: Palma, Spain, 2018; Volume 1, pp. 6888–6898. [CrossRef]
26. Singh, A.; Karayev, S.; Gutowski, K.; Abbeel, P. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale, Cambridge, MA, USA, 20–21 April 2017; L@S '17; Association for Computing Machinery: Cambridge, MA, USA, 2017; pp. 81–88. [CrossRef]

27. Keuning, H.; Jeuring, J.; Heeren, B. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, Arequipa, Peru, 11–13 July 2016; ITiCSE '16; ACM: Arequipa, Peru, 2016; pp. 41–46. [CrossRef]
28. Shute, V.J. Focus on Formative Feedback. *Rev. Educ. Res.* **2008**, *78*, 153–189. [CrossRef]
29. Gordillo, A. Effect of an Instructor-Centered Tool for Automatic Assessment of Programming Assignments on Students' Perceptions and Performance. *Sustainability* **2019**, *11*, 5568. [CrossRef]
30. Prather, J.; Pettit, R.; McMurry, K.; Peters, A.; Homer, J.; Cohen, M. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In Proceedings of the 2018 ACM Conference on International Computing Education Research, Espoo, Finland, 13–15 August 2018; ICER '18; ACM: Espoo, Finland, 2018; pp. 41–50. [CrossRef]
31. Galan, D.; Heradio, R.; Vargas, H.; Abad, I.; Cerrada, J.A. Automated Assessment of Computer Programming Practices: The 8-Years UNED Experience. *IEEE Access* **2019**, *7*, 130113–130119. [CrossRef]
32. Ju, A.; Mehne, B.; Halle, A.; Fox, A. In-Class Coding-Based Summative Assessments: Tools, Challenges, and Experience. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, Larnaca, Cyprus, 2–4 July 2018; ITiCSE 2018; ACM: Larnaca, Cyprus, 2018; pp. 75–80. [CrossRef]
33. Karvelas, I.; Li, A.; Becker, B.A. The Effects of Compilation Mechanisms and Error Message Presentation on Novice Programmer Behavior. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA, 11–14 March 2020; SIGCSE '20; ACM: Portland, OR, USA, 2020; pp. 759–765. [CrossRef]
34. Luxton-Reilly, A.; McMillan, E.; Stevenson, E.; Tempero, E.; Denny, P. Ladebug: An Online Tool to Help Novice Programmers Improve Their Debugging Skills. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, Larnaca, Cyprus, 2–4 July 2018; ITiCSE 2018; ACM: Larnaca, Cyprus, 2018; pp. 159–164. [CrossRef]
35. Yan, L.; Hu, A.; Piech, C. Pensieve: Feedback on coding process for novices. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA, 27 February–2 March 2019; Number 2 in SIGCSE 2019; ACM: Minneapolis, MN, USA, 2019; pp. 253–259. [CrossRef]
36. Benotti, L.; Martinez, M.C.; Schapachnik, F. A Tool for Introducing Computer Science with Automatic Formative Assessment. *IEEE Trans. Learn. Technol.* **2018**, *11*, 179–192. [CrossRef]
37. Brown, T.; Narasareddygari, M.R.; Singh, M.; Walia, G. Using Peer Code Review to Support Pedagogy in an Introductory Computer Programming Course. In Proceedings of the 2019 IEEE Frontiers in Education Conference (FIE), Covington, KY, USA, 16–19 October 2019; pp. 1–7. [CrossRef]
38. Loksa, D.; Ko, A.J.; Jernigan, W.; Oleson, A.; Mendez, C.J.; Burnett, M.M. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, 7–12 May 2016; CHI '16; ACM: San Jose, CA, USA, 2016; pp. 1449–1461. [CrossRef]
39. Ullah, Z.; Lajis, A.; Jamjoom, M.; Altalhi, A.; Al-Ghamdi, A.; Saleem, F. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Comput. Appl. Eng. Educ.* **2018**, *26*, 2328–2341. [CrossRef]
40. Roy, P.V.; Derval, G.; Frantzen, B.; Gego, A.; Reinbold, P. Automatic grading of programming exercises in a MOOC using the INGInious platform. In Proceedings of the European MOOC Stakeholder Summit 2015 (EMOOCs 2015), Mons, Belgium, 18–20 May 2015; pp. 86–91.
41. Restrepo-Calle, F.; Ramírez-Echeverry, J.J.; Gonzalez, F.A. Using an Interactive Software Tool for the Formative and Summative Evaluation in a Computer Programming Course: An Experience Report. *Glob. J. Eng. Educ.* **2020**, *22*, 174–185.
42. Pérez, F.; Granger, B.E. IPython: A System for Interactive Scientific Computing. *Comput. Sci. Eng.* **2007**, *9*, 21–29. [CrossRef]
43. Joshi, A.; Kale, S.; Chandel, S.; Pal, D.K. Likert Scale: Explored and Explained. *J. Appl. Sci. Technol.* **2015**, *7*, 396–403. [CrossRef]
44. Bryman, A. Qualitative data analysis. In *Social Research Methods*, 5th ed.; Oxford University Press: Oxford, UK, 2016; Chapter 24.