Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica

A.a. 2021/2022

Sessione di laurea Ottobre 2022

Sistema di Version Control per l'esplorazione nei Notebook Computazionali con progetti IoT

Relatori:

Luigi De Russis

Juan Pablo Sáenz Moreno

Candidato:

Gianluca Brezzo

Ringraziamenti

Questa tesi di laurea è il passo conclusivo del mio percorso universitario e, per questo importante traguardo, voglio ringraziare mia madre, mio padre e tutti coloro che hanno creduto in me e mi hanno sempre sostenuto.
Un ringraziamento particolare ai relatori di questa tesi, per il supporto che mi hanno fornito, per essere stati un punto di riferimento e per avermi guidato con professionalità e pazienza.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Negli scorsi anni l'importanza del mondo dell'Internet of Things, o IoT, ovvero l'estensione di internet e dei suoi utilizzi anche verso oggetti fisici, è diventata sempre più rilevante.

Alla base dell'Internet of Things ci sono i cosiddetti "oggetti intelligenti", che possono essere, oltre ai classici computer e smartphone, anche oggetti della vita quotidiana in grado di connettersi e comunicare tra di loro per poter offrire dei servizi in diversi ambiti, come quelli della sanità e della domotica.

Il progressivo diffondersi degli ambienti IoT ha portato quindi la necessità di lavorare a stretto contatto con questo mondo anche nell'ambito dei Notebook Computazionali, ovvero ambienti virtuali usati per creare documenti appartenenti al paradigma del Literate Programming, che consiste nell'intervallare frammenti più o meno grandi di codice a sezioni esplicative in un linguaggio naturale, come l'Inglese o l'Italiano.

L'utilizzo di questo tipo di Notebook permette di ottenere documenti complessi utilizzando diversi tipi di celle, che possono comprendere testo, snippet di codice, immagini e visualizzazioni grafiche, in modo da poter includere narrative complesse nel documento e presentarle in un modo più fluido e immediato (Figure 1.1 e 1.2).

Nell'ambiente del notebook è inoltre possibile eseguire il codice contenuto nelle celle, per produrre parametri intermedi che verranno utilizzati in seguito oppure per visualizzare i risultati all'interno del documento stesso.

Figura 1.1: Esempio di Notebook computazionale

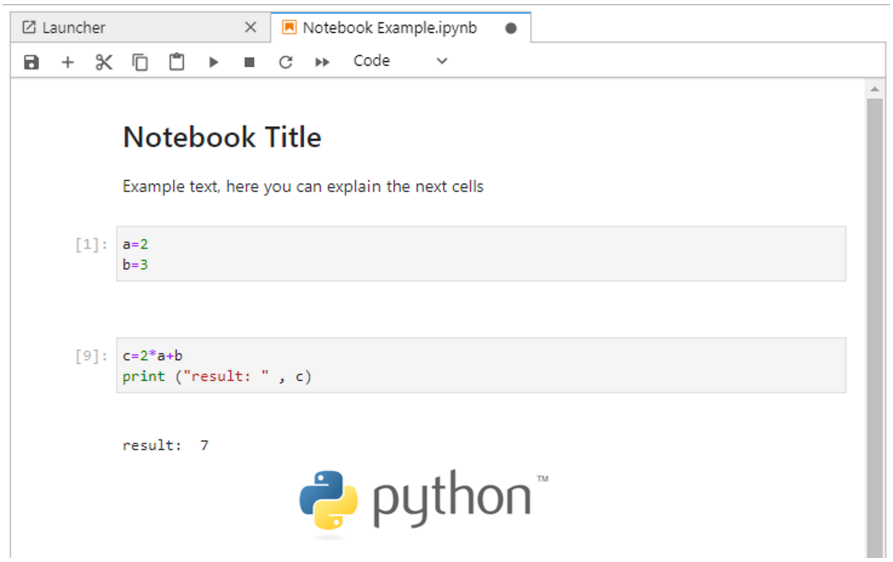
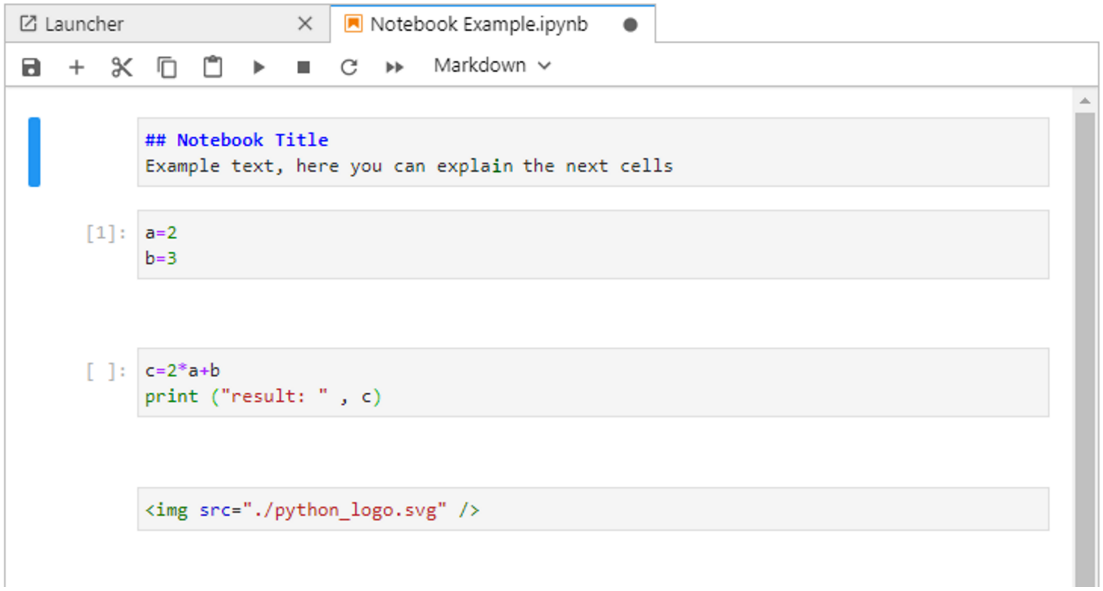


Figura 1.2: Celle di un Notebook computazionale



Lavorare con i notebook computazionali nel campo dell'IoT comporta l'avere specifiche esigenze come, tra le altre cose, il bisogno di poter usare insiemi di dati raccolti dai sistemi IoT, quello di poter controllare le versioni precedenti del documento, quello di poter verificare il processo di creazione che ha portato alla versione definitiva e quello di poter testare il codice con diversi insiemi di dati per

osservare e confrontare i risultati ottenuti.

Di conseguenza, diverse funzionalità non presenti normalmente all'interno dell'ambiente di creazione del notebook sono richieste o desiderate dagli utenti, come ad esempio un sistema di version control, che permetta di visionare e gestire versioni passate del notebook, o un modo più semplice per interagire con i dispositivi IoT all'interno di un editor di Notebook Computazionali.

I sistemi di version control più diffusi permettono di visualizzare, comparare e gestire le versioni precedenti di un file o di un insieme di file. Alcune tra le funzionalità offerte dai sistemi di versioning più diffusi, oltre alla semplice visualizzazione delle versioni e delle modifiche apportate ad ognuna di esse, sono il rollback, che premette di annullare un insieme di modifiche effettuate tornando quindi a uno stato precedente, la fork, che permette di creare un "ramo" parallelo del progetto su cui si possa lavorare indipendentemente, e la merge, una funzione complementare alla fork che permette di unire due rami del progetto, mantenendo le modifiche effettuate in entrambi, se non in conflitto tra di loro.

In mancanza di funzioni essenziali presenti negli editor di Notebook Computazionali, vengono utilizzate estensioni che possano facilitare o arricchire il processo di creazione dei documenti e successivamente semplificare quello di presentazione.

Nell'ambito del versioning, questo accade perché l'uso di un version control classico, esterno al Notebook, come può essere quello implementato da Git, comporta numerosi problemi e complicazioni, in quanto, utilizzando strumenti di versioning diffusi, come lo stesso Git, l'intero documento viene esaminato, senza permettere all'utente di constatare le effettive modifiche effettuate alle singole celle e al contenuto del notebook.

Inoltre, i notebook vengono spesso salvati come file JSON, un formato che permette di rappresentare informazioni complesse tramite una stringa strutturata che viene analizzata in fase di lettura. Questo fa sì che i potenziali conflitti generati da Git, che se presenti vengono inseriti all'interno dello stesso file di un notebook, rendano incorretta la sintassi JSON, invalidandone la struttura, e di conseguenza non permettono al file di essere aperto, facendolo quindi diventare inutilizzabile [1].

Per ovviare a questi problemi viene reso necessario adottare un sistema di version control interno all'ambiente del notebook.

1.1 Obiettivo

Questa tesi ha come obiettivo quello di progettare e realizzare meccanismi di version control per i Notebook Computazionali.

L'obiettivo verrà raggiunto tramite lo sviluppo di un'estensione che consenta l'utilizzo di funzionalità relative al version control nell'ambiente di un editor di Notebook Computazionali, in particolar modo di Jupyter Lab, per favorire la creazione, la revisione e la presentazione di notebook relativi al mondo dell'Internet of Things.

L'estensione permetterà di avere un sistema di versioning di base che permetta di controllare le versioni precedenti e di ritornare a una di esse in qualunque momento, dando in aggiunta la possibilità di scegliere il livello di granularità su cui operare le funzioni di version control: tra l'intero documento e la singola cella. L'estensione permetterà inoltre di utilizzare le funzionalità di fork e merge sul notebook.

L'interfaccia grafica dell'estensione dovrà essere immediata e dovrà permettere un utilizzo intuitivo delle varie feature anche a chi usa per la prima volta l'estensione.

Questa estensione, tramite le funzionalità offerte, avrà inoltre lo scopo di incentivare l'esplorazione (usare codice e set di dati diversi) e di facilitare l'esecuzione ripetuta di vecchi frammenti di codice ai fini di testarne i risultati, oltre all'aver un maggiore controllo sul processo che porta alla versione finale del notebook.

1.2 Struttura della tesi

La tesi è strutturata nel modo seguente:

Nel capitolo 2 si presentano i problemi e l'idea alla base della tesi, mettendo in mostra le esigenze e prendendo in esame un'estensione già rilasciata.

Nel capitolo 3 si descrivono le soluzioni impiegate per ovviare alle esigenze e ai problemi descritti in precedenza.

Nel capitolo 4 si mostrano le strutture utilizzate per la creazione dell'estensione, assieme all'interfaccia realizzata.

Nel capitolo 5 vengono evidenziati gli obiettivi raggiunti.

Nel capitolo 6 si conclude il documento e si descrivono possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

Questo capitolo descrive lo stato dell'arte e le esigenze relative al versioning per i Notebook Computazionali.

2.1 Introduzione a Jupyter

Jupyter Lab, come accennato in precedenza, è un ambiente di sviluppo e un editor per Notebook Computazionali. E' possibile utilizzare Jupyter Lab direttamente dal browser e permette di creare e modificare dei documenti computazionali, inserendo immagini, codice e parti testuali in markdown, un linguaggio di formattazione che utilizza espressioni codificate per modificare l'aspetto e l'impaginazione del testo.

L'interfaccia di Jupyter è formata da una parte centrale, contenente il corpo del documento, una sidebar, divisa in più sezioni (chiamate anche tab) che riguardano aspetti diversi come la gestione dei file relativi ai documenti, ai kernel o all'installazione delle estensioni, nonché tab relative alle funzionalità aggiunte tramite le estensioni stesse. E' presente inoltre una toolbar che racchiude i comandi relativi alla gestione e all'esecuzione delle celle.

I file relativi ad un Notebook Computazionale utilizzano, come già spiegato nello scorso capitolo, la codifica JSON per immagazzinare le informazioni contenute nel documento e i relativi metadata.

Jupyter Lab permette di utilizzare e compilare più linguaggi di programmazione

all'interno dei notebook, come Python o Javascript, e consente inoltre di aggiungerne di ulteriori tramite l'installazione di nuovi kernel, che possono andare a integrare anche altri aspetti e funzionalità e possono essere combinati con le estensioni, anch'esse installabili, tramite la tab apposita nella sidebar oppure da fonti esterne.

A loro volta le estensioni, una volta installate, possono aggiungere nuove funzionalità, migliorare l'esperienza di utilizzo di Jupyter, integrare nuovi aspetti di funzionalità esistenti, aggiungere nuovi elementi all'interfaccia grafica o modificare quelli già presenti.

Un notebook di Jupyter è composto da celle, che possono essere di tipo diverso, impostato dall'utente, a seconda della loro funzione.

Ogni cella contenente codice può essere eseguita singolarmente (in alternativa all'esecuzione dell'intero documento) e il risultato delle operazioni viene salvato internamente, potendo così essere riutilizzato per l'esecuzione delle celle successive.

Le celle, una volta eseguite, possono anche generare un output da mostrare a schermo, testuale o visivo.

2.2 Esigenze dei notebook computazionali

Da diversi paper riguardanti problemi e necessità riscontrati dagli utilizzatori di Notebook Computazionali, in particolar modo nell'ambito dell'IoT, ovvero "*The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry*" [2], "*The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool* " [3], "*Exploration and explanation in computational notebooks*" [4], "*Towards Computational Notebooks for IoT Development*" [5] e "*Fork It: Supporting Stateful Alternatives in Computational Notebooks*" [6], è emersa la necessità di poter usufruire di svariate feature, non presenti di default nell'ambiente di Jupyter, durante il processo di creazione del notebook.

Le feature più importanti e con priorità di implementazione più alta che sono emerse dai paper sono:

Branch switching: La possibilità di creare una ramificazione separata di un notebook per potere lavorare in parallelo e in modo indipendente da quella di partenza e permettere agli utenti di sperimentare nuove opzioni con più facilità, senza preoccuparsi di andare ad intaccare il lavoro svolto in precedenza. [6]

Version Control: Un'altra funzionalità pensata per incentivare l'esplorazione e la sperimentazione, direttamente collegata all'utilizzo di diversi branch, è la presenza di un sistema di Version Control, che permetta di salvare e visionare versioni passate, possibilmente anche potendone confrontare i risultati.

Campo Architectural element: Un campo che indichi a quale o a quale tipo di dispositivo IoT è collegato uno specifico notebook, in modo da poterlo riconoscere in modo immediato e modificare il comportamento in caso si cambi la tipologia di dispositivo collegato. [5]

Campo Id: Un campo univoco che permetta di identificare ogni cella e conseguentemente permettere di utilizzarle tramite riferimento in seguito. [5]

In aggiunta a queste feature, ritenute essenziali per la creazione di notebook computazionali collegati al mondo dell'IoT, ne sono emerse anche altre, contenute nella tabella sottostante insieme a quelle riportate in precedenza (Tabella 2.1):

Tabella 2.1: Feature richieste

Feature	**Priorità**
Branch switching	Alta
Version Control	Alta
Rappresentazione visiva	Opzionale
Clean-up/Function grouping	Opzionale
Campo Architectural element	Alta
Campo Id	Alta
Group linking	Opzionale
Campo di esecuzione in background	Opzionale

Oltre alle altre funzionalità richieste, nel complesso è emersa più volte e da più studi, come quelli contenuti in "*Fork It: Supporting Stateful Alternatives in Computational Notebooks*"[6] e in "*Exploration and explanation in computational notebooks*" [4], la necessità di un sistema di versioning interno al notebook stesso, che comprenda diverse delle feature citate in precedenza.

Nel dettaglio viene richiesto il poter permettere agli utenti di sperimentare con dati e frammenti di codice diversi, dando la possibilità di cambiare rapidamente la visualizzazione tra sezioni del programma passate e quelle correnti e eseguire frammenti di codice con diversi campioni di dati per confrontare gli esiti ottenuti.

In particolare, si è presentata soprattutto la difficoltà di tenere traccia del lavoro svolto e delle sperimentazioni effettuate, e conseguentemente di comprendere il percorso che ha portato alla versione finale del notebook (o di una sua parte) [3]. Questo provoca inoltre delle complicazioni nelle situazioni in cui si vuole capire il codice (e il processo creativo che ha portato a tale codice) realizzato da un altro programmatore e quelle in cui si vuole replicare la sperimentazione per intraprendere strade differenti da quelle precedenti e confrontarne i risultati.

Il sistema di versioning deve essere interno, in modo che possa quindi analizzare il contenuto effettivo delle celle e che permetta di confrontare facilmente più versioni. Questo perché i sistemi di versioning più diffusi come Git non sono in grado di fornire dettagli sul contenuto delle celle (essendo questi ultimi inclusi nei metadata del documento) e delle modifiche effettuate (la funzione di differenza non riesce ad essere efficace), facendo in modo che gli utenti che sperimentano con set di dati e codice diversi siano costretti ad usare un versioning "manuale", che nella peggiore delle ipotesi consiste nel salvataggio di più copie di un notebook rinominandole per specificare la versione (portando spesso ad avere innumerevoli duplicati dello stesso notebook con solo poche celle che differenziano l'uno dall'altro [6]), nel tenere aperte più finestre con copie dello stesso documento oppure nel mantenere più copie (spesso commentando le versioni precedenti) di una stessa cella contenenti ciascuna una versione diversa [3] [6].

Con un sistema di version control interno verrebbe anche incentivata l'annotazione delle sperimentazioni e dei relativi "vicoli ciechi", che altrimenti verrebbe spesso tralasciata, rendendo complicato il processo di tenere traccia, comparare e analizzare diverse strade e, conseguentemente, meno immediato quello di discussione di ragionamenti e risultati, soprattutto nei casi in cui le celle andrebbero riscritte e rieseguite o in cui le annotazioni dovrebbero essere rimosse per avere maggiore chiarezza e pulizia visiva all'interno del notebook [4].

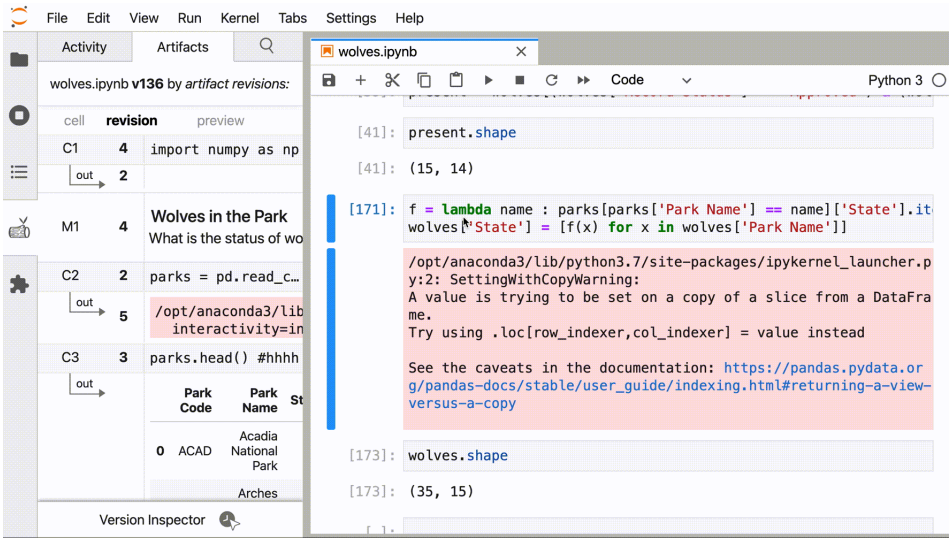
Il versioning implementato deve inoltre dare la possibilità di tornare facilmente ad una versione precedente, sempre per evitare che più versioni della stessa cella debbano essere salvate in commenti e recuperate manualmente [6].

Da altri studi è emersa la necessità di avere una funzionalità di fork per esplorare alternative che altrimenti implicherebbero la rimozione ripetuta di risultati (parziali e non) e la conseguente esecuzione delle celle e delle variabili che li producono [6].

2.3 Verdant

All'interno del paper "*The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry*" [2] viene nominata un'estensione già rilasciata, chiamata Verdant, che ha lo scopo di integrare il version control nell'ambiente dei notebook computazionali, tramite un sistema di versioning interno a Jupyter [7].

Figura 2.1: Interfaccia grafica di Verdant

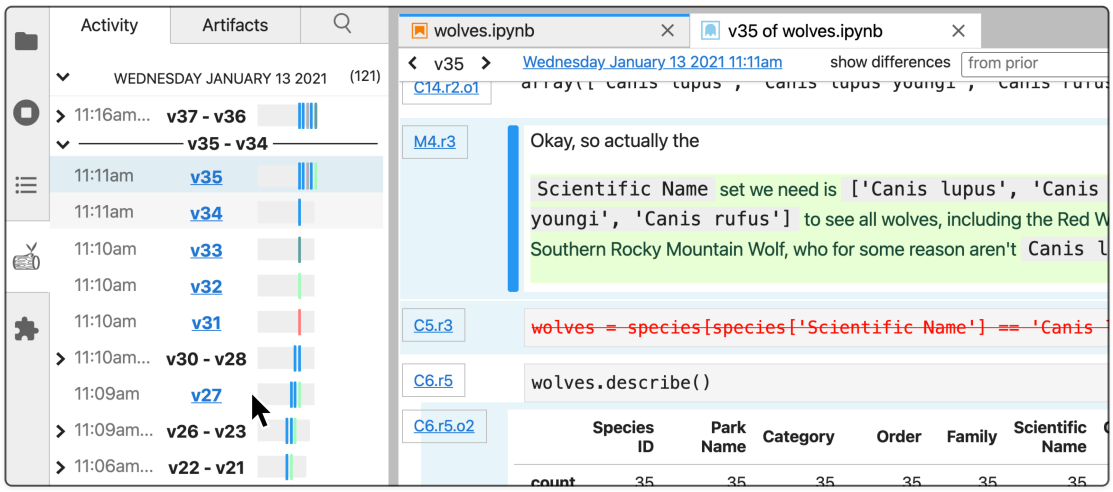


Questa estensione permette di consultare e gestire le versioni precedenti del notebook tramite una tab apposita situata nella sidebar, salvando i dati relativi al version control in un file `.ipyhistory` affiancato al file del notebook vero e proprio.

L'utente può visualizzare l'elenco delle versioni e aprire ognuna di esse in una versione in sola lettura del notebook, chiamato "ghost notebook" (Figura 2.2), per confrontare le modifiche effettuate e può cercare un frammento del notebook o una modifica specifica tramite un'apposita funzione di ricerca.

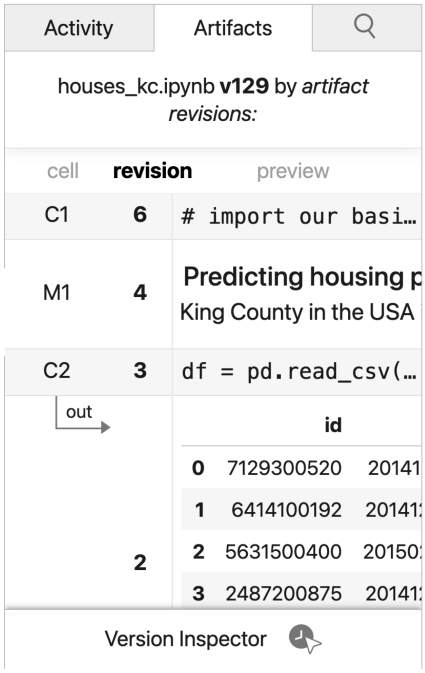
Nonostante Verdant consenta di avere una gestione profonda e dettagliata delle varie versioni, mostrando le modifiche alle celle, presenta un'interfaccia grafica non immediata e poco adatta ad utenti inesperti (Figura 2.1), con molte informazioni relative ad ambiti diverse inserite all'interno di una sola sezione dell'interfaccia grafica, sprovviste di indicazione per l'utente.

Figura 2.2: "Ghost notebook" in Verdant



Ad esempio, la tab "Artifacs", che dovrebbe aiutare a vedere l'intero insieme delle modifiche effettuate ad una cella o a una parte del notebook allo scopo di tenere traccia dei miglioramenti avvenuti nel corso del tempo, è strutturata in un modo che a prima vista può risultare poco chiaro, e può necessitare di più tentativi da parte dell'utente per capirne il funzionamento (Figura 2.3).

Figura 2.3: Tab "artifacts" in Verdant



Inoltre, l'estensione è sprovvista di alcune feature fondamentali che sono risultate essenziali dagli studi precedentemente citati, come la fork e la conseguente merge, necessarie per incentivare l'esplorazione di più percorsi, non consentendo quindi l'esplorazione simultanea di più percorsi per constatare i risultati che si otterrebbero.

Capitolo 3

Soluzione e Design

Questo capitolo descrive le feature e le esigenze selezionate e le soluzioni impiegate per ovviare a queste ultime.

3.1 Scelta delle funzionalità

Tra le varie funzionalità che si sono rivelate necessarie per i notebook computazionali legati all'Internet of Things, riportate nella tabella sottostante (Tabella 3.1), si è scelto di concentrarsi su Version Control e Branch Switching per l'obiettivo di questa tesi.

Questo non solo perché sono due feature che si sono mostrate necessarie in seguito alla conclusione di più studi, spesso indicate tra gli aspetti più importanti da avere durante la creazione dei notebook, ma perché sono anche due funzionalità legate strettamente tra di loro, in quanto la possibilità di cambiare branch all'interno di un progetto è spesso integrata nei sistemi di versioning più diffusi.

Si è optato per non focalizzarsi sugli altri due aspetti a priorità alta, ovvero il Campo id per le celle e il Campo Architectural Element per il notebook, perché, nonostante fossero due feature rilevanti, non sarebbero state abbastanza legate ad un sistema di version control da giustificarne l'inclusione nel progetto di questa tesi.

Tabella 3.1: Feature richieste

Feature	**Priorità**
Branch switching	Alta
Version Control	Alta
Rappresentazione visiva	Opzionale
Clean-up/Function grouping	Opzionale
Campo Architectural element	Alta
Campo Id	Alta
Group linking	Opzionale
Campo di esecuzione in background	Opzionale

3.2 Definizione delle caratteristiche dell'estensione

L'estensione dovrà permettere all'utente che crea e lavora con Notebook Computazionali relativi al mondo dell'Internet of Things e che utilizza insiemi di dati ricavati dai dispositivi IoT, di utilizzare un sistema di version control di base interno a Jupyter e in particolare dovrà consentire di utilizzare le seguenti funzionalità:

Scelta del tipo di version control: in seguito alla creazione di un notebook l'utente potrà scegliere se adottare un version control a livello notebook, che permette di tenere traccia delle modifiche sull'intero documento, o un version control a livello cella, che consente di avere versioni separate per ogni singola cella.

Aggiunta di una nuova versione: la funzionalità di base di un sistema di version control, che permetterà di aggiungere una nuova versione (dell'intero notebook o di una singola cella, a seconda del tipo scelto) e salvarla per utilizzi futuri.

Controllare le modifiche effettuate: l'utente dovrà essere in grado di osservare le modifiche di una versione rispetto alla precedente e confrontare le due versioni, tramite l'utilizzo di un'apposita interfaccia grafica.

Revert: l'utente dovrà essere in grado di tornare facilmente a una versione precedente per poterla testare o modificare, oltre a confrontare i risultati ottenuti con quelli di altre versioni.

Fork: l'utente dovrà poter utilizzare la funzione di fork, in modo da "sdoppiare" il notebook e di conseguenza effettuare modifiche su una delle fork create in modo indipendente dalle altre. Ogni fork avrà una sua lista di versioni, che al momento della creazione corrisponderanno a quelle che possedeva la fork di partenza. Questa funzione sarà disponibile solo per il tipo di version control a livello notebook.

Merge: funzione complementare quella di fork che permetterà di unire due fork a scelta, mantenendo le modifiche effettuate in entrambe ed mostrando i conflitti, se presenti, in modo che l'utente possa scegliere quale modifica mantenere. Così come la funzione di fork sarà disponibile solamente per il tipo di version control a livello notebook.

3.3 Scelte progettuali

3.3.1 Tipologia di version control

Come indicato in precedenza, l'estensione permetterà all'utente di decidere, in seguito alla creazione di un nuovo notebook, il tipo di version control da adottare per quel documento, scegliendo tra uno a livello notebook e uno a livello cella.

Questa seconda tipologia di version control è maggiormente indicata per i notebook in cui le celle per le quali si deve tenere traccia delle versioni sono in numero ridotto e le modifiche vengono perlopiù effettuate su singole celle. È il caso, per esempio, dei notebook narrativi in cui il numero di celle di tipo testuale o contenenti elementi grafici è di gran lunga superiore a quello delle celle di tipo codice, dove sarebbe superfluo avere un version control esteso all'interezza del documento, facendo così in modo che sia più funzionale concentrarsi sull'evoluzione di poche celle.

I due tipi di version control funzioneranno in modo simile, anche per rendere l'utilizzo di entrambi più immediato per l'utente, ma in quello a livello cella non saranno presenti le funzioni di fork e merge, in quanto aggiungerebbero complessità superflua per la ridotta quantità di codice che si andrebbe a modificare e ad analizzare.

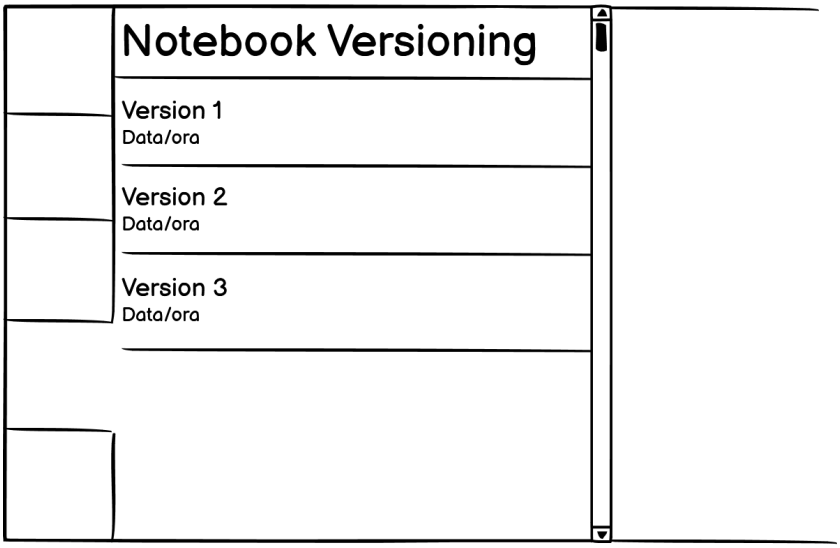
3.3.2 Aggiunta e visualizzazione di una versione

Sarà possibile effettuare la creazione di una nuova versione tramite un apposito pulsante situato nella toolbar, visualizzabile solamente una volta scelto il tipo di version control. Entrambi i tipi di versioning avranno questo pulsante, ma quello a livello cella ne avrà uno ulteriore al di sotto di ogni cella, che permetterà di aggiungere direttamente una versione senza dover selezionare precedentemente una cella, mentre il pulsante situato nella toolbar aggiungerà una versione alla cella attiva, ovvero quella correntemente selezionata dall'utente.

Se il notebook non possiede versioni verrà visualizzato un messaggio nella sidebar, nella sezione dedicata all'estensione. Una volta che una o più versioni sono state inserite da parte dell'utente, la sidebar dovrà invece mostrarne l'elenco (Figura 3.1), in un formato grafico simile a quello utilizzato da Git.

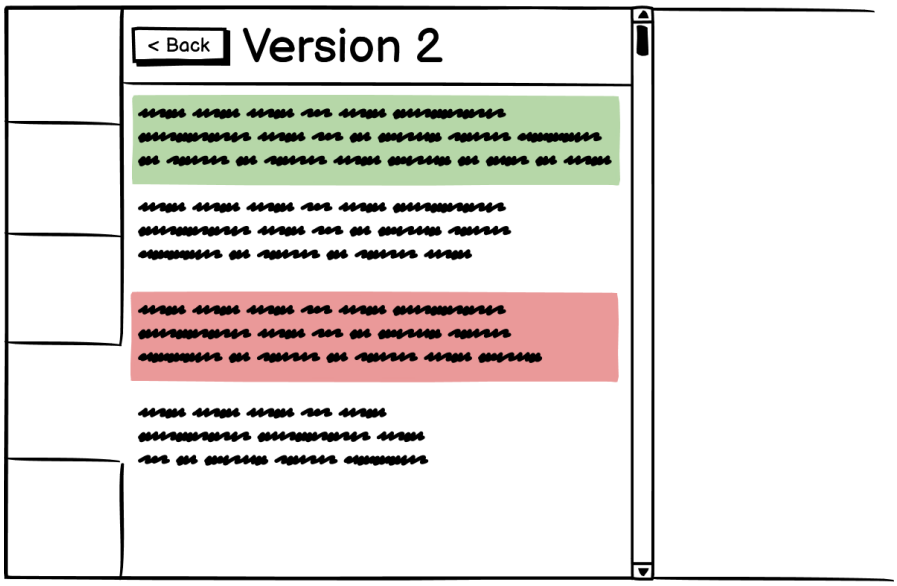
Una volta selezionata una versione, la rappresentazione deve mettere in evidenza le modifiche effettuate a ogni cella (o ad ogni riga di una cella nel caso del version control a livello cella) e permettere di confrontarle con la versione precedente (Figura 3.2).

Figura 3.1: Mockup della visualizzazione delle versioni



Per rendere più chiara e immediata la lettura delle modifiche apportate a ogni versione, seguendo lo stile di Git, verrà usato un sistema di color coding: le parti bianche saranno quelle rimaste invariate dall'ultima versione, quelle verdi quelle aggiunte e quelle rosse staranno a indicare le parti rimosse. In caso di cambiamenti

Figura 3.2: Mockup della visualizzazione dei dettagli di una versione



a una parte del notebook sarà visualizzato sia il segmento della versione precedente (in rosso) sia quello della versione corrente (in verde).

3.3.3 Revert

La funzione di revert dovrà permettere all'utente di ritornare con immediatezza al contenuto del Notebook di una versione specifica, sostituendo le celle attuali con quelle della versione selezionata, in modo da poter analizzare e confrontare i risultati ottenuti con diversi frammenti di codice. Le modifiche effettuate al Notebook non saranno salvate automaticamente, in modo da non essere definitive (a meno che l'utente non decida di salvare il documento o inserirle in una nuova versione tramite l'apposito pulsante della toolbar).

3.3.4 Fork e Merge

La funzionalità di fork deve essere programmata in modo da permettere all'utente di poter creare un nuovo insieme di versioni su cui poter lavorare liberamente senza preoccuparsi di apportare modifiche anche alla fork di partenza, in modo da incentivare la sperimentazione, soddisfacendo i bisogni indicati nei paper "*Fork it:*

Supporting Stateful Alternatives in Computational Notebooks" [6] e "*The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool*" [3].

La nuova fork sarà identica a quella di origine e le versioni corrisponderanno fino al momento della sua creazione, permettendo quindi all'utente anche di utilizzare la funzionalità di revert con le versioni precedenti, mentre le nuove versioni inserite saranno ovviamente indipendenti.

Nel momento in cui l'utente vorrà riunire due fork potrà farlo con la funzionalità di merge, tramite elementi dell'interfaccia grafica visualizzati nel caso sia già stata effettuata un'operazione di fork sul notebook. Questa funzione permetterà di riportare le modifiche effettuate su una delle due fork anche sull'altra.

La funzione di merge dovrà utilizzare un algoritmo che permetta di individuare i conflitti, ovvero modifiche alla stessa parte di codice che entrano in contrasto l'una con l'altra. In presenza di conflitti individuati dopo un'operazione di merge, questi dovranno essere mostrati all'utente, in un formato simile a quello di Git, che mostri le modifiche effettuate in entrambe le versioni all'interno di una sezione commentata, in modo che l'utente possa scegliere quale delle due parti (o una combinazione delle due) mantenere.

3.4 Architettura dell'estensione

3.4.1 Architettura di Jupyter

Dal punto di vista della creazione di un'estensione, la struttura di Jupyter è composta da tre macro-componenti: uno riguardante l'ambiente notebook vero e proprio (gestito tramite il file *factory.tsx*), uno riguardante la sidebar (gestito tramite *sidebar.tsx*) e uno riguardante la toolbar (gestito tramite il file *toolbar.tsx*). Questi tre moduli, ovvero tre file in linguaggio Typescript, un'estensione di Javascript, sono a loro volta gestiti tramite il file *index.tsx*, che carica e prepara i componenti necessari, instaurando le varie connessioni richieste.

I tre moduli interagiscono tra di loro e con il kernel, oltre a occuparsi dell'interfaccia grafica e delle interazioni che l'utente può avere con essa.

L'estensione dovrà aggiungere un nuovo pannello relativo al version control alla sidebar, che permetta di visionare le versioni e di utilizzare le funzionalità

di versioning, e inserire dei pulsanti nella toolbar, che serviranno per abilitare il version control e aggiungere nuove versioni.

Per aggiungere questi elementi nell'infrastruttura Jupyter si dovranno creare dei nuovi componenti da inserire principalmente all'interno dei moduli di sidebar e toolbar, che dovranno permettere di utilizzare le funzionalità necessarie.

Per esempio, per creare il pulsante che permetta di abilitare il version control all'interno della toolbar, si dovrà inserire un componente di tipo `ToolbarButtonComponent`, che creerà il pulsante e gestirà i comportamenti da adottare in seguito alla pressione, tramite callback.

Inoltre, per il tipo di version control a cell-level, sarà necessario un pulsante sotto ogni cella che permetta di aggiungere una versione, che dovrà essere inserito nel componente *factory*, in quanto esso si troverà nel corpo principale del notebook.

3.4.2 Interazioni tra i componenti

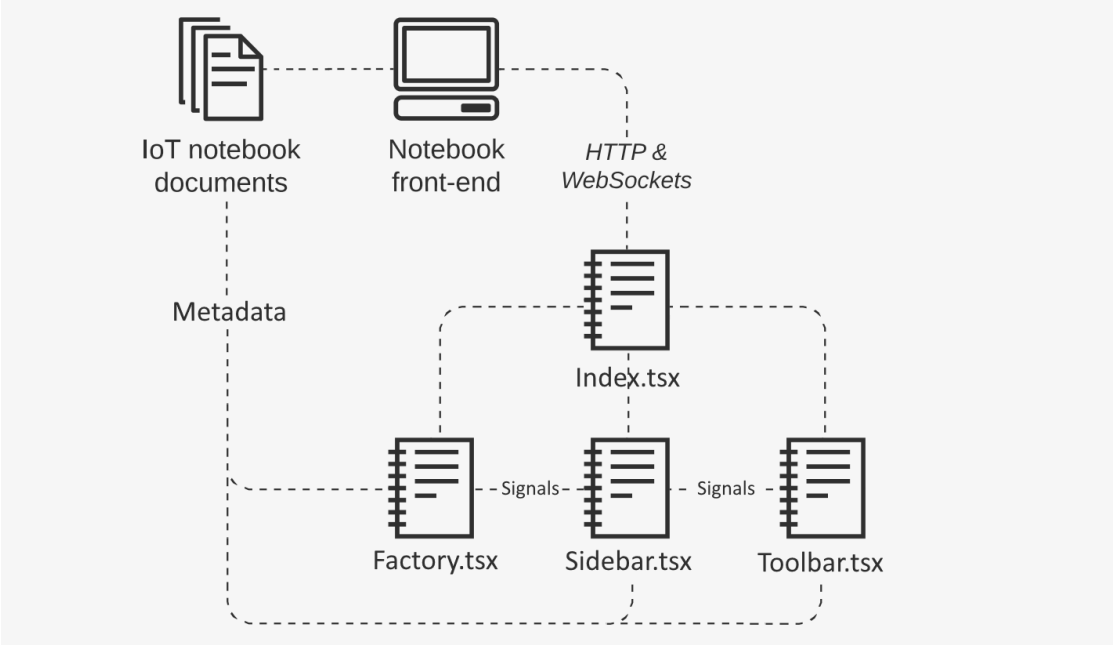
L'estensione comunicherà con l'ambiente di Jupyter tramite due modalità principali: la scrittura e lettura dei metadata di un notebook e i segnali ricevuti da altre componenti dell'ambiente di Jupyter (Figura 3.3).

La lettura dei dati riguardanti il version control avverrà tramite la funzione `JSON.parse()` della libreria, da utilizzare una volta letti i metadata, che permette di recuperare i valori anche se all'interno di strutture come vettori e matrici.

Per fare in modo che i dati visualizzati siano sempre corretti verranno utilizzati dei segnali, in particolare il segnale `services.contents.fileChanged`, che viene inviato nel momento in cui un qualunque file dell'ambiente di lavoro di Jupyter viene aggiornato, creato o eliminato, che sarà connesso alle funzioni incaricate di preparare i dati e inserirli nella sidebar per essere visualizzati dall'utente.

Alcune funzioni, come `addDocumentVersion()`, che, come suggerisce il nome, verrà utilizzata per aggiungere una funzione ad un notebook, verranno utilizzate da più parti dell'estensione e per questo dovranno essere esportate e rese adatte anche per tipi di dato diversi.

Figura 3.3: Interazione tra i componenti di Jupyter



3.4.3 React e parte grafica

All'interno del nuovo componente della sidebar vengono creati degli elementi React, una libreria Javascript open source utilizzata per la creazione di interfacce grafiche. React utilizza componenti basati su classi di tipo "stateful", ovvero che si servono di uno stato, che può essere anche propagato ad altri componenti, per decidere il loro comportamento.

In fase di render, i componenti di React creano i vari elementi di css, inizializzati con i valori opportuni, che andranno a comporre l'interfaccia grafica situata nella sidebar.

Lo stato di React viene impostato in seguito alla lettura dei metadata dal file del notebook, che a sua volta avviene successivamente alla ricezione di segnali, ovvero al cambio di notebook attivo (cioè quello mostrato a schermo al momento) o ad un cambiamento nei file utilizzati da Jupyter.

Conseguentemente al cambio di stato, sia nel caso del cambiamento del notebook attivo, sia in quello di modifiche effettuate ai file, gli elementi grafici della sidebar vengono nuovamente renderizzati in modo da contenere i dati aggiornati e corretti

per il notebook attualmente visualizzato.

Per la parte puramente estetica è invece stato utilizzato React Bootstrap [8], una versione di Bootstrap, ovvero un framework grafico utilizzato per il front end delle applicazioni web, pensato apposta per React, che implementa tutti gli elementi standard di Bootstrap come componenti di React, e permette così di usare classi css preimpostate per la visualizzazione grafica delle varie parti dell'estensione, come pulsanti e la rappresentazione delle celle.

Capitolo 4

Implementazione

Questo capitolo ha lo scopo di mostrare le strutture e le strategie utilizzate per la creazione dell'estensione di version control.

4.1 Funzionamento del version control

4.1.1 Struttura del documento

Un file di un notebook di Jupyter viene rappresentato da un file codificato in formato JSON, contenente le informazioni del documento, le varie celle e i metadata relativi a ciascuna di esse.

Per questo motivo si è scelto di basare il sistema di version control sui metadata di un file, facendo in modo di renderlo compatibile con istanze di Jupyter che non hanno l'estensione installata, permettendo in ogni caso di visionare e modificare il documento, e allo stesso tempo consentendo di avere le informazioni riguardanti le versioni semplicemente all'interno del file stesso, potendole quindi muovere da un'istanza all'altra con il semplice trasferimento del file.

All'interno del file di un notebook le celle sono rappresentate dal campo `"cells"` che, oltre al testo di ognuna di esse, conservato nel campo `"source"` riga per riga, contiene anche i relativi metadata, come la tipologia di cella e l'id (Figura 4.1).

Figura 4.1: JSON fields of cells

```
"cells": [
 {
  "cell_type": "code",
  "execution_count": 2,
  "id": "db2aebab",
  "metadata": {},
  "outputs": [],
  "source": [
   "a = 5\n",
   "b = 6\n",
   "c = 70\n"
  ]
 },
 ...
```

I metadata relativi al documento sono invece contenuti nella sezione chiamata "metadata" del modello JSON (Figura 4.2).

Figura 4.2: Document metadata

```
"metadata": {
 "kernelspec": {
  "display_name": "Javascript (Node.js)",
  "language": "javascript",
  "name": "javascript"
 },
 "language_info": {
  "file_extension": ".js",
  "mimetype": "application/javascript",
  "name": "javascript",
  "version": "14.17.0"
 },
   "ver_type": "Notebook"
}
```

4.1.2 Nuovi campi introdotti

Per permettere la creazione e la gestione delle versioni sono stati introdotti nuovi campi nel file JSON.

Durante la fase di sviluppo i campi con struttura a matrice sono stati temporaneamente implementati come insieme di singoli vettori, con un indice nel nome che ne indicasse la versione di appartenenza, in modo da rendere più semplice la fase di testing che riguardava l'ottenimento dei dati dai componenti di Jupyter, per essere poi trasformati in matrici effettive nel prodotto finale.

I nuovi campi vengono aggiunti ai metadata al momento della creazione della prima versione e nel dettaglio abbiamo:

- `"max_fork"` : il numero di fork che il documento possiede. Se è 0 non è mai stata effettuata un'operazione di fork.

- `"current_fork"` : l'indice della fork attualmente attiva e visualizzata del documento.

- `"max_doc_ver"` : un vettore contenente il numero delle versioni che lo specifico documento possiede per ogni fork.

- `"ver_cell_matrix"` : una doppia matrice contentente il testo delle celle di tutte le versioni per ogni fork.

- `"ver_color_matrix"` : una doppia matrice contentente numeri che rappresentano il colore che ogni cella di ogni versione per ogni fork dovrà assumere nella rappresentazione grafica (0 rappresenta una cella bianca, senza modifiche, 1 rappresenta una cella verde, che rappresenta l'inserzione, mentre numeri negativi indicano che una cella è stata rimossa, con il modulo del numero che rappresenta l'indice della cella nella versione precedente).

- `"ver_date_matrix"` : una matrice contenente la data e l'ora di creazione di ogni versione di ogni fork.

- `"ver_date_matrix"` : una matrice contenente il nome di ogni versione di ogni fork.

4.1.3 Scelta del tipo di version control

Successivamente alla creazione di un notebook, se l'estensione è attivata, all'utente viene indicato di abilitare il version control tramite un messaggio nella sidebar, nella tab dedicata all'estensione (Figure 4.3 e 4.4).

Figura 4.3: Schermata iniziale

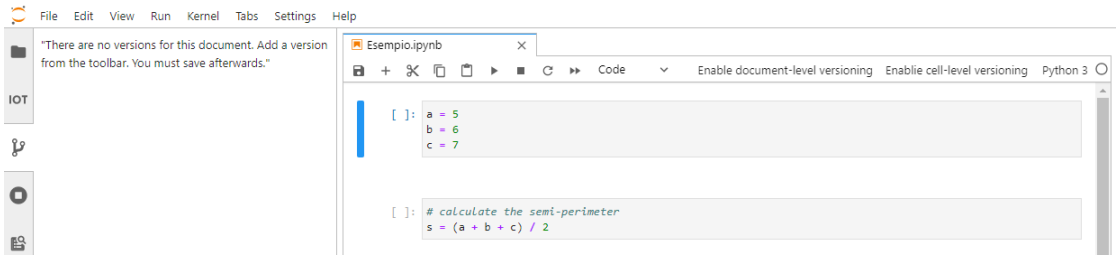
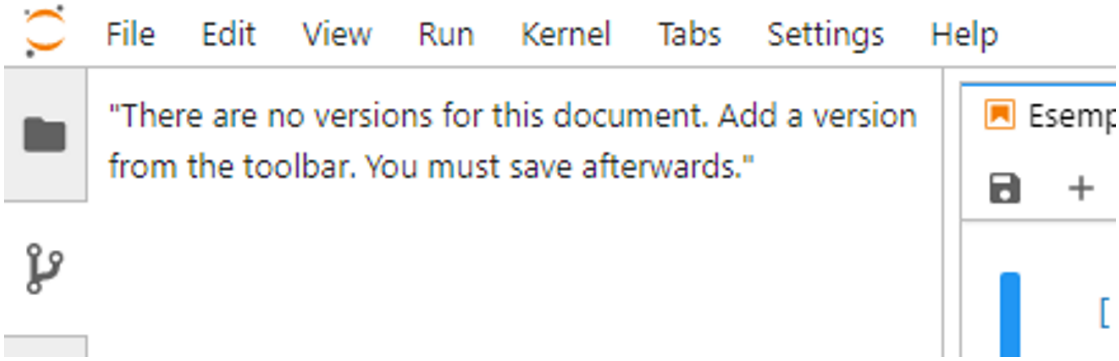


Figura 4.4: Schermata iniziale: dettaglio della sidebar

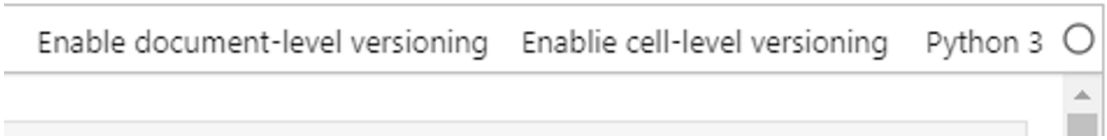


Nella toolbar l'utente può trovare due pulsanti, tramite i quali può scegliere quale tipo di version control vorrà utilizzare per il proprio documento: se quello a livello notebook o quello a livello cella (Figura 4.5).

Premendo su uno dei due pulsanti viene abilitata la possibilità di aggiungere una nuova versione al notebook e i due pulsanti vengono automaticamente nascosti, lasciando spazio a quello relativo all'aggiunta di una nuova versione.

Il tipo di version control viene salvato nei metadata del notebook nel campo `"version_type"` in modo che rimanga anche in caso di chiusura e di riapertura dello stesso. Questo campo può contenere solamente due valori: "Notebook" e "Cell", rispettivamente per i due tipi di versioning adottati.

Figura 4.5: Schermata iniziale: toolbar



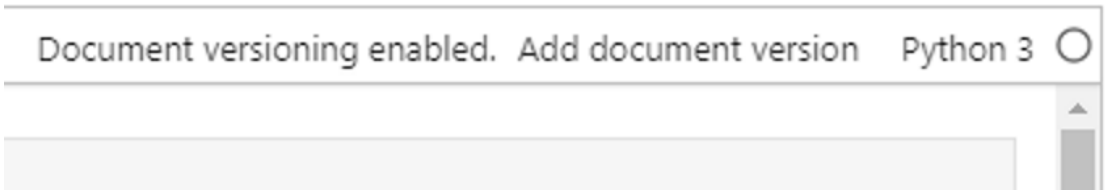
4.2 Version control a livello notebook

In questa sezione verranno dapprima analizzate le funzionalità relative al version control a livello notebook (incluse quelle relative ad entrambi i tipi, come l'aggiunta di una versione).

4.2.1 Aggiunta di una nuova versione

In seguito alla scelta del tipo di version control, l'utente può inserire una nuova versione tramite l'apposito pulsante situato nella toolbar (Figura 4.6)

Figura 4.6: Aggiunta di una nuova versione



Successivamente alla pressione del pulsante, al momento della creazione di una nuova versione, i metadata di tutte le versioni precedenti (se esistono) vengono caricati e viene aggiunto a tutte le matrici, nel vettore corrispondente alla fork attuale, un nuovo campo corrispondente al numero della versione appena creata.

Se i campi letti non esistono ne vengono creati dei nuovi con dei valori di default (poiché si presume di conseguenza che sia la prima volta che si aggiunge una versione al notebook), che verranno in seguito salvati nel file.

Nella figura 4.7 viene riportato questo processo per il campo `"maxVersion"`, ma è applicabile, con le dovute differenze richieste dal formato e dall'utilizzo di ogni campo, per ognuno di essi.

Per prima cosa, tramite la funzione `get` applicata ai metadata del notebook, viene letto il valore del campo. In seguito viene controllata che la lunghezza del vettore ottenuto non sia nulla (starebbe a significare che il campo non è stato mai inizializzato), vengono poi salvati i parametri necessari (in questo caso l'id della versione precedente e quella appena creata) e infine i campi dei metadata del notebook vengono aggiornati con i nuovi valori tramite il metodo `set`.

Figura 4.7: Lettura dei metadata

```
0  /** get the array containing the numbers of versions for each fork
       */
1  var maxVersion = []
2  if(notebook.metadata.get(MAX_DOC_VERSION)!=null){
3      maxVersion = JSON.parse(JSON.stringify((JSONExt.deepCopy(
       notebook.metadata.get(MAX_DOC_VERSION)) as PartialJSONObject)))
       ;
4  }
5
6  if (maxVersion.length!=0){
7          /** sets the old and new version number variables */
8          var oldVer = Number(maxVersion[fork])
9          var newVer = Number(maxVersion[fork]) + 1;
10         /** sets the new max version of the current fork */
11         maxVersion[fork]=newVer;
12     }
13     else{
14         /** this means it is the first version, so a new array
       must be created */
15         var newVer = 0;
16         maxVersion=[];
17         maxVersion[fork]=0;
18     }
19
20     /** updates max_doc_version metadata field  */
21     notebook.metadata.set(MAX_DOC_VERSION, maxVersion);
```

Le celle attuali vengono lette dal modello del documento (il parametro notebook) tramite il campo cells e salvate nel campo `"ver_cell_matrix"` in un nuovo indice, mentre nome, data e colori da visualizzare della nuova versione vengono generati e salvati nelle loro rispettive variabili.

Tramite il metodo `JSON.parse` si possono recuperare i dati dei vari campi direttamente con la struttura a matrice in cui sono salvati nel notebook, permettendo così di aggiungere dati in un nuovo indice senza dover ricreare la matrice da capo.

Se la versione non è la prima che viene inserita, viene chiamata la funzione `createDiff`, condivisa tra il tipo notebook level e quello cell level del version control.

Questa funzione confronta due vettori contenenti il testo della nuova e della vecchia versione per individuare le celle aggiunte (o le righe, in caso di versioning cell-level) e quelle eliminate (Figura 4.8).

Se una cella non combacia con quella della nuova versione viene ricercata in una posizione più avanzata, se viene trovata significa che il nuovo documento presenta delle celle aggiunte, in caso contrario vuol dire che la cella è stata rimossa.

Figura 4.8: createDiff()

```
0  function createDiff(oldValues: string[], newValues: string):
       number[] {
1      [...]
2       for (oldIndex = 0; oldIndex < oldSize;) {
3           if (oldValues[oldIndex] == newValues[newIndex]) {
4               /* the two cells are identical */
5               colorArray[colorIndex] = 0;
6               colorIndex++;
7               oldIndex++;
8               newIndex++;
9           }
10          else {
11              /* testing for insertion and deletion */
12              for (tempIndex = newIndex; tempIndex < newSize;
       tempIndex++) {
13                  if (oldValues[oldIndex] == newValues[tempIndex]) {
14                      /* the old cell is found further in the new
       version -> insertion */
15                      for (let t = newIndex; t < tempIndex; t++) {
16                          /* updates color array with every new cell
        */
17                          colorArray[colorIndex] = 1;
18                          colorIndex++;
19                      }
20                      /* update the new version index to be in sync
       with the old one */
21                      newIndex = tempIndex;
22                      colorArray[colorIndex] = 0;
23                      colorIndex++;
```

```
24                      oldIndex++;
25                      break;
26                  }
27              }
28              if (tempIndex == newSize) {
29                  /*the old cell is not found in the new version ->
    deletion
30                  the negative index (+1 for accounting index 0
    occurencies) is put in the color array */
31          colorArray[colorIndex] = -(oldIndex + 1);
32          colorIndex++;
33          oldIndex++;
34        }
35        else
36          newIndex++;
37      }
38    }
39    for (newIndex; newIndex < newSize; newIndex++) {
40      /* new cells added at the end of the docuemnt */
41      colorArray[colorIndex] = 1;
42      colorIndex++;
43    }
44    return colorArray;
45  }
```

L'algoritmo utlizzato dalla funzione consiste nell'effettuare una "doppia scansione" dei due vettori appartenenti alle due versioni analizzate.

Non ci sono state modifiche fino a che il contenuto dei due vettori coincide, mentre, se viene rilevata una differenza, dapprima si controlla che la cella (o la riga nel caso del version control a livello cella) non sia contenuta più avanti nella nuova versione, stando a significare che è stata effettuata una operazione di inserimento. Se invece non viene trovata allora è stata effettuata un operazione di rimozione di una o più celle (righe nel caso del version control a livello cella). Vengono prese in considerazione anche gli inserimenti "in coda" alla versione, se presenti.

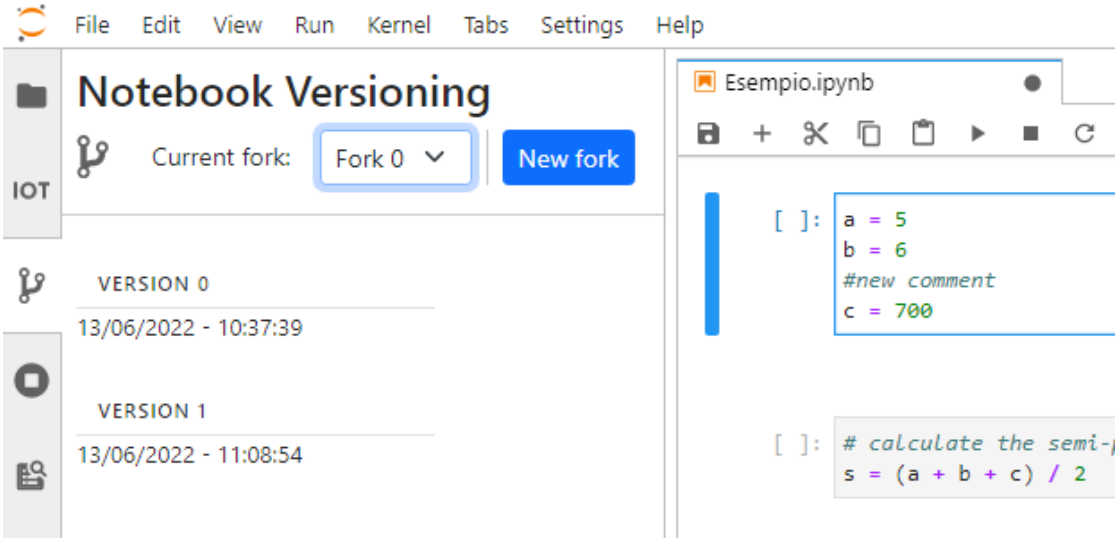
La funzione restituisce un vettore contenente i numeri che rappresentano i colori che verranno usati nella rappresentazione grafica della nuova versione.

Un'altra funzione condivisa tra i due tipi di version control è `formatDate()`, che restituisce la data e l'ora attuale nel formato desiderato, per inserirla poi nel rispettivo campo dei metadata del documento e poterla visualizzare nella sidebar sotto alla versione a cui appartiene.

4.2.2 Visualizzazione delle versioni

Una votla inserita almeno una versione in un notebook, questa verrà visualizzata in un elenco all'interno della sidebar, assieme alle altre, se presenti, con i relativi nomi e le date e le ore di creazione di ciascuna di esse (Figura 4.9).

Figura 4.9: Elenco delle versioni



Le versioni visualizzate cambieranno se l'utente passa ad un altro documento o ne apre uno nuovo, di fatto visualizzando le versioni relative solo al notebook in primo piano.

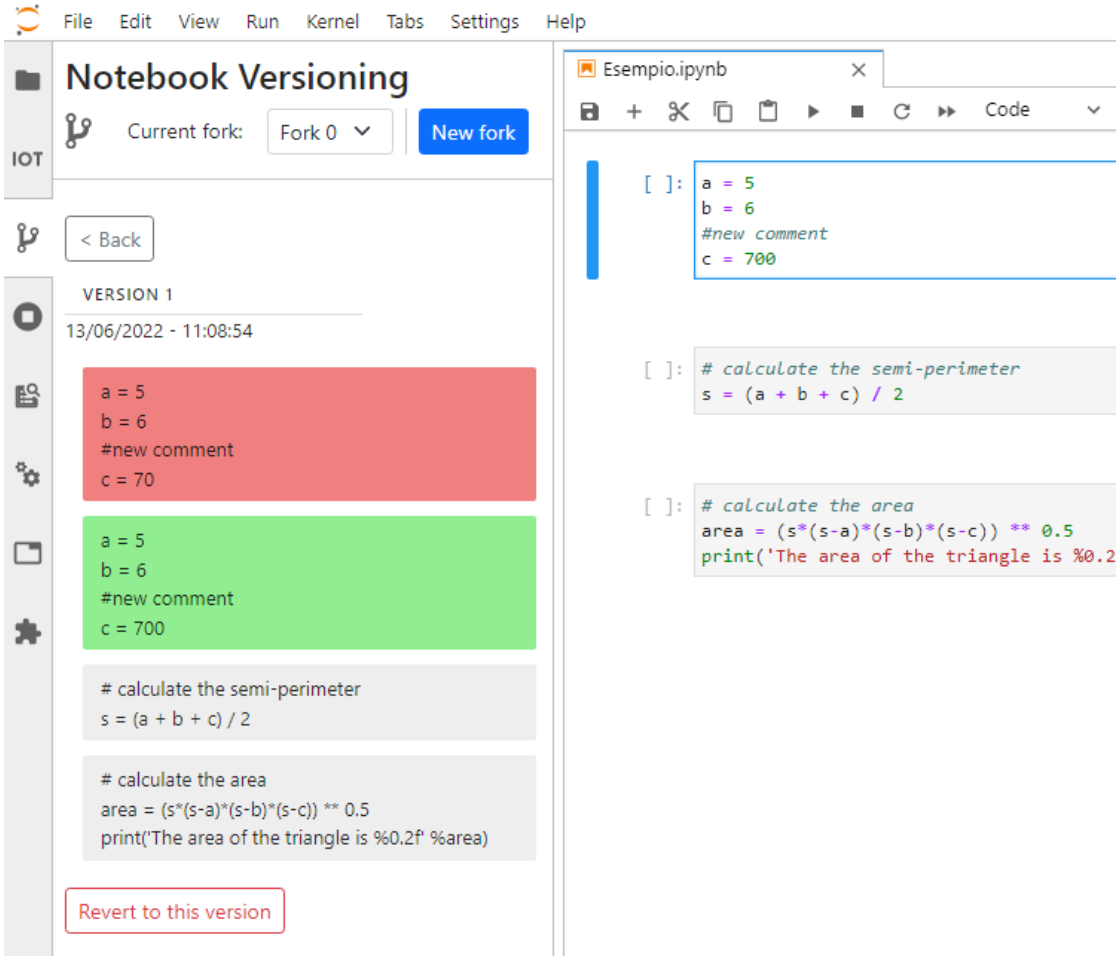
Cliccando su una singola versione verranno mostrate le celle relative, mettendo in evidenza le differenze tra quella versione e quella precedente tramite una visualizzazione grafica (Figura 4.10). Da questa schermata è possibile tornare all'elenco delle versioni tramite l'apposito pulsante *Back*.

Come proposto in fase di progettazione, lo stile della rappresentazione è simile a quello di Git e i colori di ogni cella rappresentano le eventuali modifiche apportate al documento: una cella bianca indica che non ha subito modifiche dalla versione precedente, una verde indica che è stata aggiunta e una rossa indica la sua rimozione.

Nel caso di modifica ad una cella compariranno sia la cella prima della modifica (in rosso) sia quella attuale (in verde), come mostrato in Figura 4.10.

Se quella visualizzata è la prima versione, tutte le celle saranno rappresentate

Figura 4.10: Dettagli di una versione



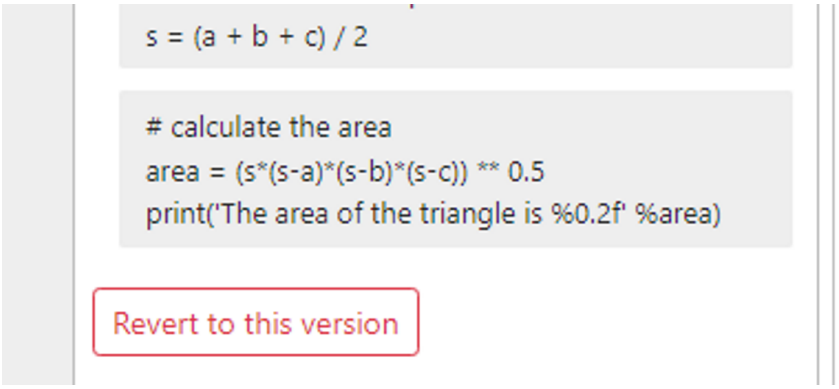
in verde, in quanto tutte sono state aggiunte in seguito alla creazione del notebook e prima di quella della versione.

4.2.3 Funzione di revert

Come mostrato in precedenza, una delle funzionalità inserita nell'estensione di versioning è quella di tornare a una versione precedente del notebook (o della singola cella, in caso di versioning a livello cella). Per effettuare tale azione, al momento della pressione del pulsante *Revert to this version*, situato nella sidebar, alla fine delle informazioni di una versione (Figura 4.11), viene chiamata la funzione

`revertToVersion()` (Figura 4.12), rispettivamente all'interno degli oggetti di tipo NotebookLevelVersions e CellContent.

Figura 4.11: Pulsante di revert



La funzione Notebook-level, tramite i parametri passati all'elemento di react, recupera il vettore celle del documento attualmente visualizzato dal NotebookTracker e le celle della versione a cui si vuole tornare (scremando le celle corrispondenti alla versione precedente, ovvero quelle rappresentate da un numero negativo nel vettore dei colori), per poi, se necessario, creare ulteriori celle o rimuovere quelle in eccesso, sostituendo il contenuto visualizzato del notebook con quello della versione a cui si sta tornando.

Figura 4.12: revertToVersion()

```
0  function revertToVersion(){
1      [...]
2      /** number of "red" cells, removed from previous version*/
3      let redCells=0;
4      for(let i=0; i<oldLength; i++){
5          if(props.color[i]=="red"){
6              redCells++
7          }
8      }
9      /** gets the actual number of old document cells and creates
       new cells if necessary */
10     for(let i=0; i<oldLength-redCells-currentLength; i++){
11         props.tracker.currentWidget.model.cells.push(
12             props.tracker.currentWidget.model.contentFactory.
       createCodeCell({}));
13     }
14     /** replaces cell content with content from old version*/
15     let cellIndex=0;
16     for(let i=0; i<oldLength; i++){
```

```
17          if(props.color[i]!="red"){
18              props.tracker.currentWidget.model.cells.get(cellIndex)
    .value.text=props.cells[props.fork][props.index][i]
19              cellIndex++;
20          }
21      }
22      /** removes cells if necessary */
23      props.tracker.currentWidget.model.cells.removeRange(cellIndex,
    props.tracker.currentWidget.model.cells.length+1)
24  }
```

Il corrispettivo di questa funzione per il modello di versioning cell-level è notevolmente più semplice, in quanto si deve solamente sostituire il contenuto di una singola cella, che viene salvato come una semplice stringa nei metadata del notebook, senza bisogno quindi di calcolare il numero di celle da aggiungere o rimuovere (Figura 4.13).

Figura 4.13: revertToVersion()

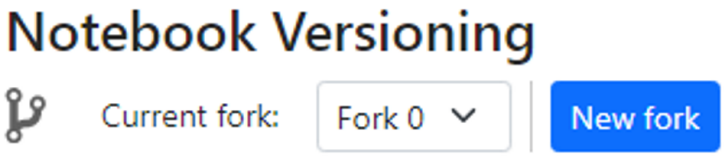
```
0  function revertToVersion(){
1      let cells = JSON.parse(JSON.stringify((JSONExt.deepCopy(props.
    tracker.currentWidget.model.metadata.get("ver_cell_matrix")))))
    ;
2      props.tracker.currentWidget.model.cells.get(props.cellNumber).
    value.text=cells[props.cellNumber][props.versionNumber];
3  }
```

4.3 Fork e merge

4.3.1 Creazione nuova fork

Come deciso nei capitoli precedenti, se il version control selezionato per il documento è quello di tipo notebook level, è possibile effettuare la funzione di fork sul documento, tramite il pulsante *New fork*, situato nella sidebar, in alto nella lista delle versioni di un notebook.

Figura 4.14: Gestione Fork



Nel momento in cui l'utente decide di aggiungere una nuova fork tramite il pulsante apposito, viene chiamata la funzione `addFork()`, che ha il compito di copiare le versioni e i relativi dettagli presenti in quel momento nella fork di partenza in una nuova fork, e aggiungerli poi in ogni matrice dei metadata, inserendoli nei rispettivi campi.

Una particolarità di questa funzione è la creazione della matrice `unsaved`, che verrà usata nel momento in cui l'utente deciderà di cambiare la fork attualmente visualizzata (Figura 4.14). Questa matrice contiene una copia temporanea delle celle del notebook, che vengono aggiornate prima di ogni cambiamento della fork attuale e reinserite nel notebook quando l'utente ritornerà alla rispettiva fork, in modo che eventuali modifiche non salvate vengano ripristinate.

Figura 4.15: Matrice unsaved nella funzione addFork()

```
0  function addFork(){
1      [...]
2
3      /** stores the current cells into unsaved matrix, to be
       restored in case of fork switching */
4      unsaved[oldFork]=currentCells
5      /** unsaved cells of new fork will be the same as the version
       itself */
6      unsaved[newFork]=cells[oldFork][oldVer];
7      props.tracker.currentWidget.model.metadata.set('unsaved',
       unsaved);
8
9      [...]
10 }
```

4.3.2 Cambio della fork attuale

Se l'utente decide di cambiare la fork attuale, tramite il menu a dropdown affiancato al pulsante *New fork*, viene invocata la funzione `changeFork()`.

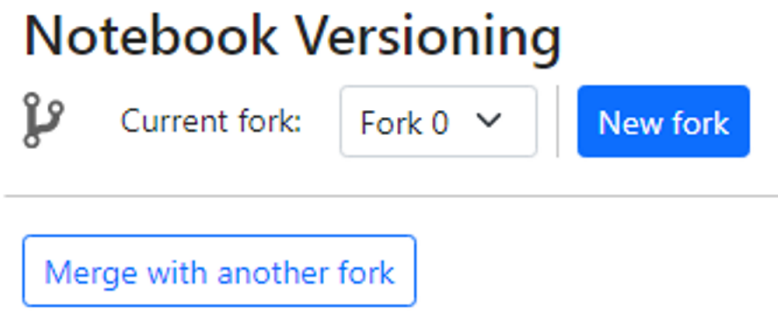
La funzione ha il compito di aggiornare le celle del notebook (e conseguentemente i metadata), salvando però, in modo analogo alla funzione `addFork()`, le celle presenti al momento del cambio nel campo unsaved, in modo che non ci siano perdite di cambiamenti se queste ultime non sono state ancora inserite in una versione. Per la sostituzione delle celle si utilizza un metodo analogo a quello utilizzato nella funzione di revert.

Inoltre, questa funzione deve anche cambiare gli elementi di react che sono visualizzati nella sidebar, in modo da mostrare solo quelli relativi alla fork attuale.

4.3.3 Merge

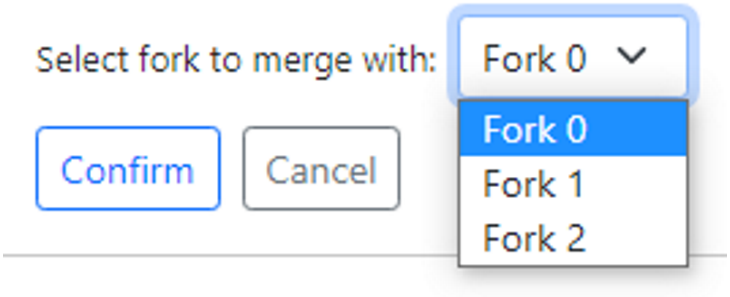
Se sul notebook è stata effettuata almeno una volta la funzione di fork, apparirà il menu riguardante la funzionalità di merge nella sidebar. L'utente può effettuare l'operazione di merge tramite il pulsante *Merge with another fork*, presente immediatamente sotto al menu di fork (Figura 4.16).

Figura 4.16



Una volta che l'utente ha selezionato tramite il menu a tendina (Figura 4.17), nella lista del quale ovviamente non sarà presente la fork che si sta visualizzando attualmente, la fork con cui effettuare l'operazione di merge con quella attuale, la funzione `mergeForks()` viene chiamata (Figura 4.19).

Figura 4.17: Scelta della fork



Questa funzione utilizza metodi della libreria diff-match-patch di Google [9], che utilizza un sistema di merge di tipo "three-way", ovvero la comparazione di due diramazioni con il loro punto di origine.

Per ottenere le celle che verranno usate come "base", il punto di origine del sistema diff-match-patch, ogni versione delle due fork viene confrontata, partendo dalla prima e fermandosi all'ultimo paio di versioni in cui tutte le celle delle due fork sono identiche (Figura 4.18).

Figura 4.18: Individuazione della versione di base

```
0  function mergeForks () {
1      [...]
2      /** left = cells from target fork */
3      left = cells[targetFork];
4      /** right = cells from current fork*/
5      right = cells[currentFork];
6      let minLength = right.length;
7      if(left.length < right.length)
8          minLength=left.length
9      /**calculates the last version in which the two forks are the
       same */
10     let baseLength=-1;
11     while(JSON.stringify(left[baseLength])===JSON.stringify(right[
       baseLength]) && baseLength<minLength){
12         baseLength++;
13     }
14     baseLength--;
15     base=cells[currentFork][baseLength];
16     [...]
17 }
```

Una volta individuata la versione di base, la funzione provvede ad aggiornare i metadata del documento, rimuovendo la fork di partenza e spostando tutti i campi delle fork successive all'indice precedente del vettore, in modo da rimuovere tutti gli indici relativi alla fork rimossa, eliminando quindi i "buchi" che altrimenti rimarrebbero nelle matrici.

4.3.4 Gestione dei conflitti

Non avendo la libreria diff-match-patch una gestione dei conflitti e volendo evitare la rimozione non desiderata di modifiche da uno dei due branch, la soluzione che si è sviluppata prevede la creazione di due risultati intermedi, utilizzando per entrambi una diversa fork come "master", ovvero quella che vedrà le sue modifiche prevalere in caso di conflitto (Figura 4.19).

Figura 4.19: Utilizzo della funzione di merge

```
0  function mergeForks () {
1      [...]
2
3      /** creates empty strings to be filled with JSON content */
4      let newJson1 = "";
5      let newJson2 = "";
6
7      /** in order to let the user choose which side to pick
8       * in case of conflict, the google diff_match_patch
9       * library has to be run twice, once between left and base
10      * and once between right and base
11      */
12     newJson1 = createMerge(base, left, right)
13     newJson2 = createMerge(base, right, left)
14
15     [...]
16 }
```

L'algoritmo di ricerca dei conflitti è simile a quello utilizzato per individuare i cambiamenti tra due versioni.

Una volta ottenuti i due risultati, questi vengono comparati, dapprima cella per cella. Se le due celle presentano differenze si cerca una corrispondenza più avanti nel documento, e, se queste non vengono trovate, si ripete il processo sulla singola cella

confrontando riga per riga. Se neanche in questo caso si trova una corrispondenza si ottiene infine un risultato che conterrà entrambe le versioni, separate da commenti, in modo che l'utente possa scegliere quale mantenere in caso di conflitto.

Il testo contenuto nei commenti che verranno inseriti in caso di conflitti cambia a seconda del tipo di kernel utilizzato, in modo che vengano effettivamente riconosciuti come tali una volta visualizzati nelle celle (Figura 4.20).

Figura 4.20: Tipi di commento

```
0  /** gets notebook language in order to decide which type
1   * of comment to adopt
2   */
3  let comment="//"
4  if(props.tracker.currentWidget.model.defaultKernelLanguage=="
       python")
5      comment="#"
6
7  /** strings used to indicate which fork the conflict side belongs
       to */
8  let leftStr="Fork "+targetFork;
9  let rightStr="Fork "+currentFork;
10
11 [...]
12
13 for(let x=j; x<k; x++){
14     fnalMerge[finalMergeIndex]=comment+leftStr+"->\n\n"+comment+"
       ----------\n"+newJson2[x]+"\n"+comment+"<-"+rightStr+"\n";
15     finalMergeIndex++;
16     j++;
17 }
```

All'interno dei commenti viene anche indicato il numero della fork a cui appartiene ciascuna parte del conflitto, in modo che l'utente possa riconoscerle facilmente e in modo immediato.

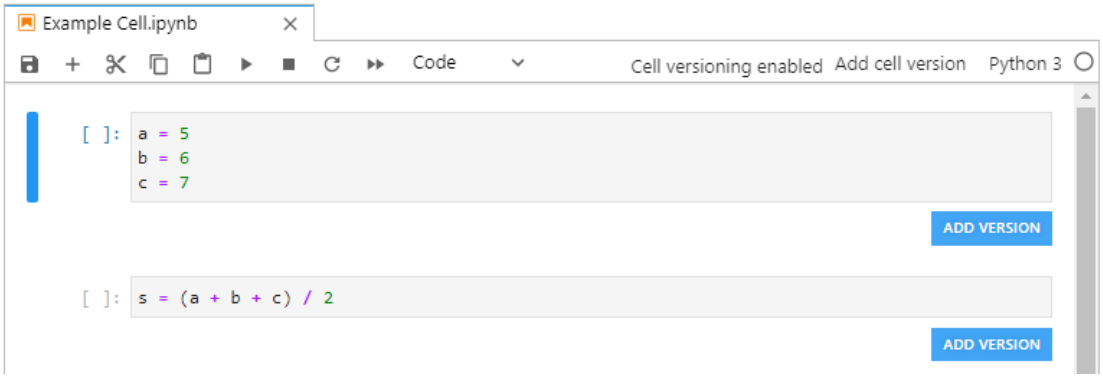
Una volta che l'operazione di merge è completata e gli eventuali conflitti sono stati individuati e gestiti tramite i commenti, la funzione aggiorna i campi dei metadata della fork di destinazione, inserendo le nuove celle ottenute poco prima, in modo analogo a quanto avviene al momento dell'inserimento di una nuova versione.

4.4 Version control a livello cella

Il tipo di version control a livello cella riprende le funzionalità di aggiunta versione, visualizzazione delle varie versioni e revert (che però opera solamente sulla singola cella) dal tipo a livello notebook. Come già indicato in precedenza non sono invece presenti le funzioni di fork e merge.

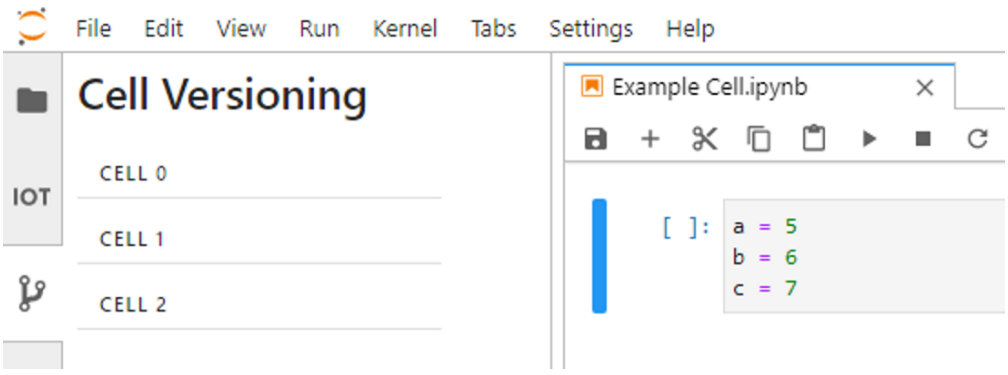
Inoltre, nel caso in cui l'utente selezioni questo tipo di version control, un pulsante per aggiungere una versione apparirà al di sotto di ogni cella (Figura 4.21). In alternativa a questo pulsante l'utente può utilizzare anche quello situato nella toolbar, che aggiunge una versione alla cella attiva, ovvero quella attualmente selezionata al momento dell'aggiunta della versione.

Figura 4.21: Pulsante aggiuntivo al di sotto delle celle

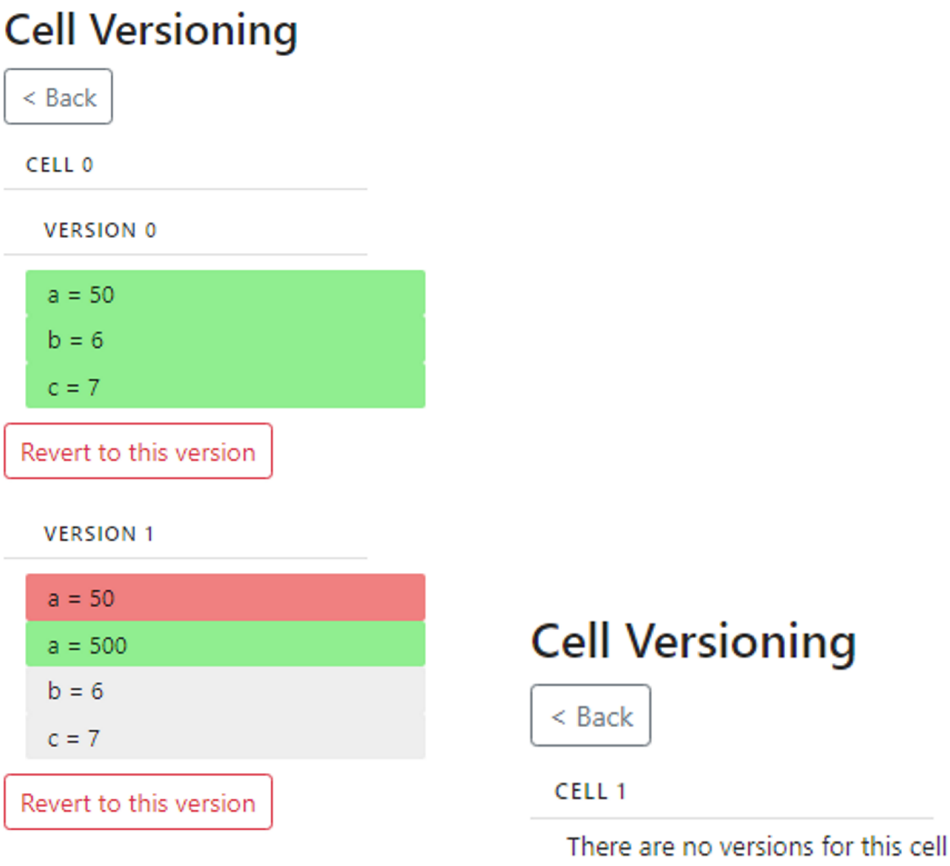


Al momento dell'apertura di un notebook, nella sidebar verrà visualizzato l'elenco delle celle che il documento possiede (Figura 4.22).

Figura 4.22: Elenco delle celle



Cliccando su una cella dalla lista nella sidebar, se questa ha delle versioni ne verrà visualizzato l'elenco (Figura 4.23a), nel caso contrario verrà comunicato all'utente che la cella non ha versioni (Figura 4.23b).



(a) Cella con versioni **(b)** Cella senza versioni

Figura 4.23: Versioni di una cella

L'elenco delle versioni visualizzate sarà quello corrispondente alla cella che l'utente avrà selezionato e si aggiornerà ogni volta che cliccherà su un'altra cella, indicando l'assenza di versioni se quest'ultima non ne possiede ancora.

4.4.1 Campo "max_version"

La differenza principale a livello di metadata tra il sistema di versioning document-level e quello cell-level consiste nel campo `"max_version"`, in quanto

quest'ultimo non sarà solamente un singolo numero per ogni versione, ma un vettore contente la massima versione per ogni cella.

4.5 Aggiornamento della sidebar

4.5.1 Riconoscimento del documento aperto

Per fare in modo che la sidebar rifletta il documento attualmente visualizzato dall'utente, il componente react VersioningComponent, aggiunto alla sidebar, utilizza un oggetto di tipo NotebookTracker, che permette di inviare segnali in caso di aggiunta o aggiornamento di un widget (un sovrainsieme che contiene anche i notebook aperti) o nel momento in cui il widget in primo piano (che corrisponde a `tracker.currentWidget`) venga sostituito con un altro (Figura 4.24).

Figura 4.24: Segnali del NotebookTracker

```
0 tracker: NotebookTracker;
1
2 [...]
3
4 this.tracker.currentChanged.connect(this.componentDidMount, this);
5 this.tracker.widgetAdded.connect(this.componentDidMount, this);
6 this.tracker.widgetUpdated.connect(this.componentDidMount, this);
```

I segnali vengono connessi alla funzione `componentDidMount()`, che viene chiamata conseguentemente al lancio di uno di essi e contiene al suo interno la funzione `populate()`, che si occupa di inizializzare i dati necessari per la visualizzazione grafica (Figura 4.25).

Figura 4.25: componentDidMount()

```
0 componentDidMount() {
1         this.populate();
2 }
```

4.5.2 Aggiornamento della sidebar

Nel momento in cui l'utente aggiunge una versione tramite la toolbar (o il pulsante sotto la cella stessa nel caso del version control di tipo cell-level) i metadata del notebook vengono aggiornati. Ogni volta che questo avviene, alla fine dell'operazione il file corrispettivo viene aggiornato e un segnale, emesso dall'oggetto services di tipo FileBrowserModel, provoca la chiamata della funzione `componentDidMount()` (Figura 4.26).

Figura 4.26: fileChanged

```
0 const { services } = this.model.manager;
1 services.contents.fileChanged.connect(this.componentDidMount, this
    );
```

La funzione `populate()` ottiene i dati necessari alla visualizzazione e li inserisce nella toolbar. Essendo all'interno di componentDidMount(), il cambiamento di stato di react provoca l'immediato aggiornamento della toolbar.

La funzione opera nel seguente modo (Figura 4.27): ottiene il titolo del documento attualmente aperto tramite il NotebookTracker passato come parametro all'estensione della sidebar, crea le variabili che saranno utilizzate per contenere i metadata richiesti e controlla il tipo di version control, se presente, entrando poi in uno dei due costrutti if, a seconda del dato letto.

Figura 4.27: populate()

```
0  populate(){
1      [...]
2
3      /* get the current widget, if it exists then gets its title*/
4      let notebook = this.tracker.currentWidget;
5      if(this.tracker.currentWidget){
6          this.title = this.tracker.currentWidget.title.label;
7      }
8      /*force a refresh of the file model */
9      this.props.model.refresh;
10
11     /*scans the files and checks if the current notebook is in the
       file model*/
12     toArray(this.props.model.items()).forEach(item => {
```

```
13          if (item.type === 'notebook' && item.name === this.title)
    {
14              this.contents.get(this.title).then(() => {
15                  /*gets the versioning type of the notebook, if it
    exists*/
16                  if (typeof notebook.model.metadata.get("ver_type")
     != undefined) {
17                      this.type = String (notebook.model.metadata.
    get("ver_type"));
18                  }
19                  /*Notebook-level versioning*/
20                  if(this.type =="Notebook"){
21                      [...]
22                  }
23                  /*Cell-level versioning*/
24                  if(this.type =="Cell"){
25                      [...]
26                  }
27                  /*triggers the update of the sidebar even if the
    other state parameters did not change*/
28                  let triggerStateChange=!this.state.newState;
29                  this.setState({ type: this.type, title: this.title
    , names: this.versionNames, dates:this.versionDates, color:
    this.color, cells: this.cells, newState: triggerStateChange});
30              });
31          }
32      })
33  }
```

Per fare in modo che la sidebar sia sempre aggiornata, anche nel caso in cui gli altri parametri dello stato non siano cambiati, viene impostato un nuovo stato con un parametro booleano, `newState`, ottenuto negando il valore precedente e che conseguentemente cambia ogni volta.

La struttura delle due opzioni è molto simile, la principale differenza è data dall'assenza delle fork nel version control a livello cella e la conseguente mancanza di doppie matrici per i metadata del Notebook.

Entrambe hanno lo scopo di riempire le matrici che saranno utilizzate per la rappresentazione grafica nella sidebar, controllando se il notebook possiede o meno una versione (invece del solo tipo di version control, già inserito nei metadata al momento dell'attivazione con uno dei due pulsanti nella toolbar) tramite il campo `"max_doc_version"`, per poi ottenere i metadata tramite parse JSON e inserire il contenuto delle celle (già predisposte per l'ordine di visualizzazione, che

comprende quindi anche le celle rimosse dalla versione precedente) e i nomi delle classi css di ogni cella che saranno usati in seguito.

Per preparare le due matrici contenenti il testo e il colore delle celle da visualizzare, si legge il campo relativo ai colori di quella versione, andando a recuperare anche le celle rimosse dalla versione precedente, da inserire successivamente nella lista delle celle mostrate, in caso di indici negativi (Figure 4.28).

Figura 4.28: Creazione delle matrici cells e color nella funzione populate()

```
0  /*outer for, cycles through forks*/
1  for(let k=0; k<=maxFork; k++){
2      this.color[k]=[];
3      this.cells[k]=[];
4      /*cycles through each version of the fork*/
5      for(let i=0; i<=this.maxVer[k]; i++){
6          this.color[k][i]=[];
7          this.cells[k][i]=[];
8          let index=0;
9          /*cycles through each cell of the version*/
10         for(let j=0; j<tempColors[k][i].length; j++){
11             /*prepares the matrix containg the css class names for
       each cell*/
12             if(tempColors[k][i][j]==0){
13                 this.color[k][i][j]='container cellversion white';
14                 this.cells[k][i][j]=tempCells[k][i][index];
15                 index++;
16             }
17             else if (tempColors[k][i][j]==1){
18                 this.color[k][i][j]='container cellversion  green'
      ;
19                 this.cells[k][i][j]=tempCells[k][i][index];
20                 index++;
21             }
22             else{
23                 this.color[k][i][j]='container cellversion red';
24                 this.cells[k][i][j]=tempCells[k][i-1][(-(
      tempColors[k][i][j]))-1];
25             }
26         }
27     }
28 }
```

La funzione per il versioning a livello cella è simile, ma deve ripetere il processo per ogni singola riga di ogni cella di ogni versione, aggiungendo quindi un ciclo più

interno al for.

Successivamente viene chiamata la funzione `render()` (Figura 4.29), che ha il compito di creare i vari componenti di react. A seconda del tipo di versioning e della presenza o meno di versioni effettive verranno creati componenti diversi, secondo le esigenze. Ad esempio, se non sono presenti versioni, ma è solamente abilitato il versioning, non saranno creati, e quindi nemmeno renderizzati, i componenti che altrimenti conterrebbero le varie versioni.

Questo non rappresenta un problema neanche in caso della creazione della prima versione di un Notebook, perché ogni modifica al documento, e conseguentemente al suo file, provoca l'invio di un segnale, che a sua volta fa in modo che venga chiamata la funzione `populate()` e venga effettuato il successivo re-rendering dell'interfaccia dell'estensione nella sidebar.

Figura 4.29: render()

```
0  /**Notebook-level version, for safety checks that the length of
       the cells matrix is the same as the number of forks of the
       notebook. */
1  if(this.type=='Notebook'){
2      /*the notebook has one version and is not only version-enabled
       */
3      if(this.cells.length==Number(this.tracker.currentWidget.model.
       metadata.get("max_fork"))+1){
4          /*No forks, doesn't creates the merge-related components
       */
5          if(Number(this.tracker.currentWidget.model.metadata.get("
       max_fork")) ==0){ [...] }
6          /*The notebook is forked, creates also the merge-related
       components*/
7          else{ [...] }
8      }
9      /*The notebook is not versioned, only versioned-enabled*/
10     else{ [...] }
11 }
12 /*Cell-level versioning*/
13 else if(this.type=='Cell'){ [...] }
14 /*Type is not defined*/
15 else{
16     /*Try populate function again*/
17     if(this.cells.length!=0){
18         this.populate();
19     }
20     /**if after another run of the populate function the type is
       still not defined, then the notebook is not versioned */
```

```
21      return (
22          <div className="merge-margin-left merge-margin-right merge
    -margin-top">
23              "There are no versions for this document.
24              Add a version from the toolbar. You must save
    afterwards."
25          </div>
26      );
27  }
```

Dentro ogni return, a seconda della condizione soddisfatta, verranno creati i vari componenti css che andranno a costituire l'interfaccia grafica della sidebar.

Questi componenti saranno renderizzati con una struttura ad albero, dove i componenti più interni verranno creati a loro volta da quelli più esterni. Nel caso delle versioni o della lista delle celle, i componenti saranno renderizzati all'interno di un ciclo for, servendosi dei dati precedentemente raccolti dalla funzione `populate()`, che verranno utilizzati per riempire i vari elementi delle liste.

Capitolo 5

Risultati

Questo capitolo descrive i risultati ottenuti al termine della Tesi.

5.1 Obiettivi raggiunti

L'obiettivo generale della tesi è stato quello di sviluppare un'estensione per Jupyter che permettesse di utilizzare le funzioni base di un sistema di version control.

Questo obiettivo è stato pienamente raggiunto in quanto l'estensione consente di consultare le versioni precedenti di un notebook, di effettuare le operazioni di revert, fork e merge e di scegliere la granularità del versioning, sull'intero documento o sulle singole celle, a seconda del tipo di notebook che si sta usando.

L'estensione è completamente integrata nell'ambiente di Jupyter ed è indipendente da altri sistemi di version control, andando quindi a risolvere i problemi citati in precedenza, che altrimenti si avrebbero utilizzando strumenti esterni, ovvero l'impossibilità di controllare i cambiamenti effettuati direttamente tramite i sistemi di version control esterni e i conflitti che si genererebbero erroneamente a causa della struttura dei file dei notebook e verrebbero mostrati dagli strumenti di versioning.

Per quanto riguarda invece l'obiettivo di avere un'interfaccia grafica immediata, nonostante avere una valutazione sul campo sarebbe stata l'opzione più ideale, in modo da testare con un gruppo di persone quanto l'user interface sia effettivamente

semplice e intuitiva da usare, la componente visiva è stata realizzata con lo scopo di essere quanto più chiara possibile anche durante il primo utilizzo dell'estensione.

Rispetto a Verdant, infatti, la user interface si presenta in modo più lineare e guidato, indicando all'utente i passi da effettuare, oltre che a essere divisa in sezioni ben distinte (lista delle versioni, dettagli di una singola versione...) e conseguentemente contenenti meno informazioni alla volta, in modo che siano più facilmente leggibili e navigabili da parte dell'utente, in contrapposizione alla quantità elevata di informazioni racchiuse da ogni schermata di Verdant.

Le varie sezioni possiedono inoltre meno elementi superflui visualizzati a schermo e utilizzano una struttura più semplice per rappresentare le informazioni, servendosi del color coding per differenziare i vari tipi di modifica effettuati al notebook (o alla cella, nel caso del version control cell-level) in ogni versione.

L'insieme di questi due risultati rende indirettamente conseguito anche il terzo obiettivo, ovvero quello di incentivare il processo di esplorazione e sperimentazione durante la creazione dei notebook (anche se pure in questo caso lo scenario ideale si otterrebbe testando questo aspetto tramite una valutazione sul campo), in quanto permette rapidamente di eseguire una parte di codice relativa a una versione precedente, di intraprendere nuove possibili strade per il notebook in una fork separata senza preoccuparsi di quella principale e di confrontare più versioni dello stesso notebook andando a evidenziare le differenze.

Capitolo 6

Conclusioni

Questo capitolo conclude la Tesi e presenta alcuni possibili sviluppi futuri.

6.1 Conclusioni

Il lavoro che è stato svolto nell'ambito di questa tesi ha permesso di raggiungere gli obiettivi fissati in partenza, cioè realizzare un'estensione che permettesse di utilizzare funzioni di version control per incentivare l'esplorazione e la sperimentazione nei notebook computazionali legati al mondo dell'Internet of Things.

Nonostante il mondo dell'IoT si stia diffondendo sempre di più, dopo un'analisi dei paper citati in precedenza sono emersi bisogni specifici relativi all'utilizzo dei notebook computazionali, ma non sono emersi strumenti che presentino tutte le caratteristiche richieste.

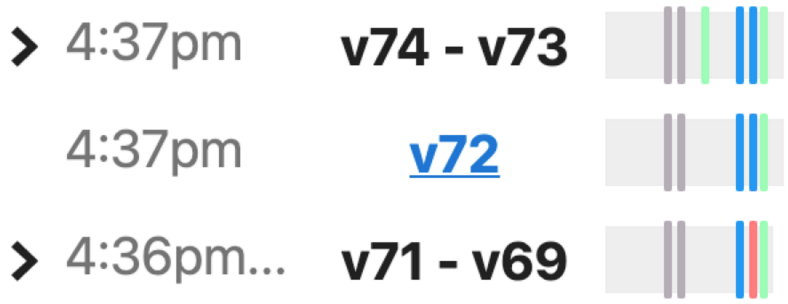
L'estensione è stata infine sviluppata avendo come obiettivo principale la realizzazione di un'interfaccia grafica immediata e semplice da capire anche al primo utilizzo, ma che non andasse a impedire l'uso delle funzionalità richieste.

6.2 Lavoro futuro

Il primo passo da considerare prima di possibili sviluppi futuri è sicuramente quello di effettuare un test sul campo applicato su un gruppo di persone, in modo da evidenziare le componenti più utili ed eventuali criticità che sarebbero messe in risalto tramite l'utilizzo dell'estensione per un periodo di tempo più o meno esteso.

Un'altra possibile direzione da intraprendere potrebbe essere quella di un ulteriore miglioramento dell'interfaccia grafica, che, oltre a un perfezionamento a livello estetico, comprenda ad esempio una visualizzazione grafica simile a quella presente in Verdant, che mostri in che punto del documento sono state effettuate delle modifiche in una singola versione ancor prima di vederne i dettagli, tramite una barra contenente linee di colore diverso a seconda del tipo di modifica, ma rielaborata in modo da essere più immediata, con un rapporto tra colore della barra e tipo di modifica più chiaro da comprendere.

Figura 6.1: Visualizzazione modifiche in Verdant



Un'ulteriore funzionalità che si può riprendere dall'estensione già esistente è quella di poter aprire una (o più di una) versione passata del notebook in una finestra "fantasma", in sola lettura, in modo da poterne visualizzare i segmenti di notebook a cui l'utente è interessato senza dover necessariamente utilizzare la funzione di revert.

Inoltre, si potrebbe aggiungere una gestione delle fork a "albero", che possa cioè mostrare la fork di partenza da cui ne è stata creata una specifica e permetta di utilizzare la funzione di merge un livello alla volta unendo una fork alla sua fork di partenza, fino ad arrivare, eventualmente, alla prima fork del notebook.

Bibliografia

[1] Jeremy Howard. *The Jupyter+git problem is now solved*. https://www.fast.ai/2022/08/25/jupyter-git/. Ago. 2022 (cit. a p. 3).

[2] Sam Lau, Ian Drosos, Julia M. Markel e Philip J. Guo. «The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry». In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pp. 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201 (cit. alle pp. 8, 11).

[3] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John e Brad A. Myers. «The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool». In: (apr. 2018), pp. 1–11 (cit. alle pp. 8, 10, 20).

[4] Adam Rule, Aurélien Tabard e James D Hollan. «Exploration and explanation in computational notebooks». In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12 (cit. alle pp. 8–10).

[5] F. Corno, L. De Russis e J.P Saenz. «Towards Computational Notebooks for IoT Development». In: (mag. 2019), pp. 1–6 (cit. alle pp. 8, 9).

[6] Nathaniel Weinman, Titus Drucker Steven M. abd Barik e Robert DeLine. «Fork It: Supporting Stateful Alternatives in Computational Notebooks». In: *CHI '21: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (mag. 2021), pp. 1–12 (cit. alle pp. 8–10, 20).

[7] Mary Beth Kary. *Verdant*. https://github.com/mkery/Verdant. 2021 (cit. a p. 11).

[8] *React Bootstrap*. https://react-bootstrap.github.io/ (cit. a p. 23).

[9] Neil Fraser. *diff-match-patch*. https://github.com/google/diff-match-patch. Lug. 2019 (cit. a p. 39).