Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

DIDATTICA DELLA PROGRAMMAZIONE: ANALISI QUANTITATIVA DI COMPORTAMENTI ED ERRORI DEI NOVIZI IN UN CORSO INTRODUTTIVO

Relatore:
Chiar.mo Prof.
Michael Lodi

Presentata da:
ALESSANDRO FREDA

Correlatori:
Dott. Stefano Pio Zingaro
Dott. Marco Sbaraglia

Sessione I
Anno Accademico 2020-2021

Abstract

Imparare a programmare è difficile. La programmazione richiede un grande sforzo della memoria di lavoro e la conoscenza approfondita della parte teorica. Questo per gli studenti novizi può essere molto complesso. In letteratura sono tanti gli studi che evidenziano le principali difficoltà cognitive che rendono il processo dell'apprendimento della programmazione particolarmente difficile. Il compito dei docenti è quello di evitare che gli studenti creino modelli mentali sbagliati e, nel caso in cui questi si fossero creati, correggerli in maniera tempestiva in modo tale da rendere la didattica della programmazione più efficace. Concetti come la ricorsione e l'istanziazione degli oggetti, ad esempio, risultano essere difficili da comprendere in particolare per un principiante che è in grado di pensare ai programmi in termini poco astratti. Per questo motivo risulta essere di grande importanza il ruolo del docente, che è in grado di stimolare ed incentivare lo studente alla creazione di modelli mentali corretti attraverso l'applicazione, ad esempio, del tracciamento del codice e la visualizzazione della macchina concettuale. Mediante l'analisi di dati raccolti durante un corso a distanza di introduzione alla programmazione in Python per non informatici si è cercato di estrarre informazioni utili al fine di correlare i comportamenti degli studenti durante le esercitazioni di programmazione con i loro risultati nel corso. In prospettiva, i dati sono stati letti alla luce delle principali difficoltà cognitive che l'apprendimento della programmazione pone al fine di proporre possibili strategie da adottare per superarle. L'analisi ha mostrato che gli studenti con un maggiore numero di eventi dei log sono gli studenti che hanno avuto i risultati migliori all'esame. Esercitarsi è, dunque, un passaggio fondamentale per l'apprendimento della programmazione.

Indice

Elenco delle tabelle

Elenco delle figure

Introduzione

L'informatica sta rimodellando il mondo che ci circonda a un ritmo sempre crescente, sta cambiando il modo in cui lavoriamo, scopriamo, comunichiamo, impariamo, creiamo, ci divertiamo e tanto altro ancora. Essendo diventata una disciplina estremamente pervasiva nella vita di tutti noi è nata conseguentemente la necessità di rendere l'informatica una materia di studio sin dalla più giovane età.
La ricerca sulla didattica dell'informatica ha svolto importanti progressi solo a partire dalla seconda metà del secolo scorso. I ricercatori nel campo della didattica dell'informatica hanno studiato, in particolare, come gli studenti imparano a programmare e quali sono i principali problemi riscontrati durante il processo di apprendimento della programmazione. Imparare a programmare è difficile. I corsi di introduzione alla programmazione spesso hanno un alto tasso di fallimento e scarsi risultati per gli studenti. A differenza dei programmatori novizi, i programmatori esperti sono bravi a riconoscere, utilizzare e adattare modelli e schemi nella programmazione. Sono più veloci e più precisi e sono in grado di attingere a un'ampia gamma di esempi, fonti di conoscenza e strategie efficaci. Al contrario, sono numerose le difficoltà cognitive che gli studenti novizi possono avere durante la fase di apprendimento della programmazione. Infatti, i problemi con i quali uno studente, che impara per la prima volta la programmazione, ha a che fare sono tanti. Le scienze cognitive negli anni hanno fortemente influenzato le metodologie e gli approcci di insegnamento dell'informatica. Ad esempio il concetto di modello mentale nell'ambito della programmazione assume grande rilevanza. Sviluppare modelli mentali corretti, è necessario per comprendere al meglio i concetti della programmazione ed applicarli correttamente durante la risoluzione di problemi. Gli studenti, sono infatti portati a creare delle idee errate, chiamate misconcezioni, che determinano incompren-

sioni e fallimenti nel processo di apprendimento. Inoltre, la programmazione richiede un notevole utilizzo della memoria di lavoro. La programmazione, inoltre, necessita di un grande sforzo cognitivo. Il carico cognitivo nella programmazione, infatti, è un fattore determinante nella fase di apprendimento.
La seguente tesi può essere strutturata in due parti. La prima parte tratta della didattica della programmazione e delle principali difficoltà cognitive della programmazione degli studenti novizi. La seconda parte tratta dell'analisi descrittiva dei dati raccolti durante il corso a distanza di introduzione alla programmazione in Python per studenti dell'università di Bologna del corso di studi di Matematica. I dati raccolti corrispondono ai log che l'ambiente di sviluppo Thonny ha generato durante le lezioni di laboratorio di programmazione. Thonny è un ambiente di sviluppo per Python progettato specificamente per i novizi. Circa 100 studenti hanno acconsentito alla raccolta dei log. L'analisi è stata effettuata con l'obiettivo di estrarre informazioni utili al fine di correlare i comportamenti degli studenti durante le esercitazioni di programmazione con i loro risultati nel corso e di comprendere il loro comportamento durante le principali fasi di apprendimento. Sono state individuate alcune caratteristiche dei log raccolti capaci di fornire utili informazioni. I tipi di errori, il numero di volte in cui gli studenti hanno eseguito il codice e il numero di volte in cui gli studenti hanno incollato il codice nell'ambiente di sviluppo sono alcune delle caratteristiche che i log hanno evidenziato che sono risultate interessanti per questo tipo di analisi. Inoltre, essendo i log per loro natura delle sequenze di eventi registrati nel tempo, è stato possibile analizzare la progressione degli studenti durante le giornate di laboratorio.

Capitolo 1

L'informatica come materia di studio e di ricerca

Oggi l'informatica ha assunto un ruolo fondamentale nella società. In tale contesto chi si occupa di insegnare informatica ha il grande e importante compito di insegnare alle nuove generazioni questa affascinante disciplina scientifica. Nel seguente capitolo, dopo una breve introduzione sull'importanza dell'insegnamento dell'informatica, verrà trattata in sintesi la storia della didattica dell'informatica. L'informatica è una scienza molto recente. Dunque, rispetto a quello che avviene in altre scienze come la Matematica e la Fisica, in cui da secoli la didattica disciplinare è oggetto di studio e di ricerca, la ricerca in didattica dell'informatica affonda le sue radici a partire dagli anni '60 del secolo scorso dal momento in cui i ricercatori iniziarono a raccogliere dati e studiare il processo di apprendimento della programmazione. Verranno successivamente trattate le tre principali teorie dell'apprendimento: comportamentismo, cognitivismo e costruttivismo. Per concludere verrà poi trattato un argomento molto vasto che è quello della scienza cognitiva, ma riassunto ed applicato alla didattica della programmazione. Il testo di riferimento utilizzato nel seguente capitolo è [1].

1.1 Importanza della didattica dell'informatica (e della programmazione)

Oggi è un pensiero comune sull'informatica ritenere che le nuove generazioni definite come *nativi digitali*, ovvero la generazione di persone nate e cresciute con le tecnologie digitali, abbiano intrinsecamente sviluppato competenze logiche e matematiche data la loro abilità nell'utilizzo di strumenti digitali. Saper utilizzare uno strumento tecnologico non significa necessariamente comprendere i princìpi alla base del funzionamento dei sistemi e della tecnologia informatica: non basta saper guidare una macchina per comprendere il funzionamento di un motore a scoppio! Dunque, in un mondo dove la tecnologia diventa sempre più incisiva e pervasiva nella società e allo stesso tempo assume di giorno in giorno sempre più rilevanza, è importante educare le nuove generazioni a comprendere il loro funzionamento e a sviluppare capacità di problem solving in modo creativo ed efficiente. Gli studenti, infatti, devono essere in grado di imparare ad utilizzare i computer per far sì che questi risolvano problemi. Ciò è possibile anche attraverso la programmazione in un contesto di gioco. La didattica dell'informatica in tale contesto assume una grande rilevanza, infatti se il compito della scuola è anche quello di formare cittadini capaci di pensare autonomamente insegnare l'informatica, oggi, è un passaggio inevitabile.

Nel mondo l'informatica sta diventando sempre di più un'importante materia di studio e di ricerca. L'ex presidente degli Stati Uniti Barack Obama nel 2016 ha stanziato centinaia di milioni di dollari per "Computer Science for All", un'iniziativa che parte dalla scuola materna fino al liceo al fine di introdurre, migliorare e potenziare la didattica dell'informatica nelle scuole. Dal punto di vista sociale stiamo vedendo sempre di più come l'economia sta cambiando rapidamente, i lavori stanno cambiando continuamente e sia gli studiosi che i leader di grandi aziende stanno riconoscendo sempre di più che l'informatica è una nuova scienza necessaria per nuove ed emergenti opportunità economiche e sociali. Anche le più famose aziende tech del mondo oggi fanno a gara per insegnare la programmazione dispiegando ingenti quantità di soldi per sostenere progetti sulla didattica della programmazione.

Sono tanti, dunque, i motivi per il quale l'informatica oggi ha assunto un valore ed un'im-

portanza sempre maggiore anche in contesti educativi. In [1] vengono individuate, anche a partire dall'analisi di importanti lavori presenti nella letteratura come quelli di diSessa [2] e Wing [3], quattro principali considerazioni oggetto di discussione sull'importanza della didattica dell'informatica :

- **Il mercato del lavoro**: uno dei principali motivi per il quale l'informatica negli ultimi anni ha assunto un valore sempre più importante anche nella didattica è anche grazie al cambiamento del mercato del lavoro. Questa logica è principalmente correlata all'incremento esponenziale della necessità di lavoratori con nuove competenze. Inoltre, come accennato in precedenza, queste dinamiche sono spesso sostenute dai leader del settore e dai leader politici. Sono numerose le richieste di lavoro nel campo dell'informatica e questo numero aumenterà negli anni a venire, con la data science e l'intelligenza artificiale che diventeranno i settori principali ed estremamente rilevanti nell'economia mondiale del futuro. Si sostiene, infatti, che il motore del successo delle economie mondiali sarà anche dovuto alla capacità dei sistemi scolastici ed educativi a generare sempre più persone con una buona preparazione nella materie scientifiche ed ingegneristiche.

- **Il pensiero computazionale**: il secondo argomento di discussione sull'importanza della didattica dell'informatica riguarda il concetto del *pensiero computazionale*, esposto in [3] da Jeannette Wing. Wing ha evidenziato l'importanza del pensiero computazionale come capacità di un individuo di risolvere i problemi e comprendere il comportamento umano, attingendo dai principi fondamentali dell'informatica. Wing sostiene che il modo di pensare, e altre strategie di risoluzione di problemi tipici dell'informatica siano universalmente importanti, anche in altre discipline.

- **L'alfabetizzazione computazionale**: altro tema di dibattito sull'importanza della didattica è quello della *computational literacy* (che potremmo tradurre come "alfabetizzazione computazionale", da non confondere però con alfabetizzazione digitale). Il principale sostenitore di questo tema è Andrea diSessa, che, nel suo libro [2], spiega come l'alfabetizzazione computazionale sia diversa da quello che Wing chiama pensiero computazionale. L'autore afferma che l'alfabetizzazione computazionale non è semplicemente una nuova abilità che costituisce la base per nuove

opportunità lavorative o, più in generale, nuovi approcci di risoluzione dei problemi ispirata all'informatica, ma è un insieme di elementi materiali, cognitivi e sociali che generano un nuovo modo di conoscere, pensare, apprendere e rappresentare la conoscenza. Una vera e propria nuova alfabetizzazione, che renderà possibili nuovi tipi di operazioni e processi mentali della conoscenza che cambierà il modo in cui le persone interagiscono tra loro e utilizzano tecnologie digitali.

- **L'equa partecipazione**: ultimo argomento di dibattito tratta dell'equa partecipazione allo studio dell'informatica. Gli studenti esclusi dalla didattica dell'informatica potranno avere difficoltà a partecipare pienamente alla società del ventunesimo secolo. Risulta evidente come la conoscenza dell'informatica sarà sempre più necessaria per il lavoro e per risolvere problemi che le sfide del progresso tecnologico pone. Deve essere dunque anche dovere di un paese rendere la didattica dell'informatica disponibile a chiunque, in quanto l'informatica diventerà sempre più cruciale anche per la partecipazione civica e il processo decisionale. Importante è dunque rendere i futuri cittadini consapevoli degli strumenti tecnologici che utilizzano, sapere ad esempio cosa sono gli algoritmi, e come questi potenti strumenti possono manipolare e distorcere la realtà che ci circonda. Dimostrazione di quanto detto fino ad ora sono tutti i movimenti che oggi rappresentano l'ultima frontiera del complottismo irrazionale nati da "fake news" sui social network.

1.2 Storia della didattica dell'informatica

Rispetto ad altre scienze come la Matematica e la Fisica, in cui la didattica disciplinare è oggetto di studio e di ricerca da molti secoli, la ricerca sulla didattica dell'informatica ha svolto importanti progressi solo a partire dalla seconda metà del secolo scorso. I ricercatori nel campo della didattica dell'informatica hanno studiato, in particolare, come gli studenti imparano a programmare e come sia eventualmente possibile migliorare il processo di apprendimento della programmazione. Linguaggi di programmazione come Fortran (1957) e COBOL (1959) furono originariamente inventati per essere più facili dell'assembly al fine di rendere la programmazione disponibile al maggior numero possibile di programmatori. Successivamente linguaggi di programmazione come BASIC

(1964) e Pascal (1970) furono inventati esplicitamente per facilitare l'apprendimento della programmazione Infatti fu proprio alla fine degli anni '60 che i ricercatori iniziarono a raccogliere dati e studiare il processo di apprendimento della programmazione. In quegli anni c'erano due categorie di studiosi sulla didattica dell'informatica, i quali seppure trattassero dello stesso argomento, avevano obiettivi diversi.
La prima categoria di ricercatori era concentrata sullo studio della programmazione come attività da comprendere in termini psicologici. In quegli anni l'informatica si è sviluppata come tecnologia diffusa e realtà commerciale, rendendo la figura del programmatore molto ricercata. Infatti prima della nascita delle qualifiche accademiche, la maggior parte dei programmatori veniva formata internamente a carico dei propri datori di lavoro. Lo scopo degli studiosi era quindi quello di comprendere, a partire da uno studio teorico ed empirico della programmazione come abilità umana, le dinamiche cognitive legate all'apprendimento. Tra gli studiosi è importante ricordare Gerald Marvin Weinberg, che illustrò tali concetti nel suo libro [4], primo libro sulla psicologia della programmazione informatica. Il libro di Weinberg, nonostante sia piuttosto breve, contiene risultati empirici sia sull'apprendimento della programmazione che sull'abilità della programmazione. Il suo lavoro fornì spunti molto interessanti per la ricerca. Infatti alcune delle principali caratteristiche delle metodologie di ricerca odierne sono emerse allora, come il confronto fra principianti ed esperti su diversi aspetti come quello dell'abilità a programmare e l'analisi della facilità di apprendimento di diverse rappresentazioni del codice. La seconda categoria di studi era incentrata più specificamente sull'apprendimento della programmazione, e mediante la programmazione. Emblematico è il lavoro di Papert e dei suoi colleghi sul linguaggio di programmazione Logo [5]. L'obiettivo di questi studiosi era quello di comprendere i benefici cognitivi della programmazione per gli studenti che si approcciano ad essa. Logo, un linguaggio derivato da LISP, fu ideato da Papert e colleghi con l'obiettivo di insegnare la matematica e la risoluzione dei problemi attraverso la programmazione, piuttosto che insegnare la programmazione in sé. Logo era un ingranaggio tecnologico nella visione di Papert per il processo educativo. Lo scopo era quello di insegnare attraverso la programmazione.
Inoltre è utile ricordare che in quegli anni l'attenzione era molta di più su ciò che i programmi potevano fare piuttosto che sulla forma del linguaggio di programmazione stesso.

Un altro parametro da tenere in considerazione è il fatto che il linguaggio fu progettato alla fine degli anni '60, quando la psicologia della programmazione era solo agli inizi. Inoltre è bene ricordare che in quegli anni erano due le aree di lavoro che risultavano essere di particolare interesse per gli studiosi: le difficoltà dei novizi con la programmazione e la progettazione di linguaggi di programmazione per i novizi. Problemi che verranno trattati nei capitoli successivi.
Le scienze cognitive, emerse tra gli anni '70 e '80, con l'utilizzo di metodologie di ricerca basate sulla modellazione della cognizione umana ebbero un grande impatto sulla ricerca della didattica dell'informatica. Negli anni '80 nacquero due gruppi internazionali per promuovere e sostenere la ricerca sulla didattica dell'informatica. Uno era il Psychology of Programming Interest Group (PPIG), formato nel Regno Unito nel 1987. L'altro era l'Empirical Studies of Programmers (ESP), formato negli Stati Uniti. Dalla metà degli anni '70 all'inizio degli anni '90, la didattica dell'informatica ha acquisito sempre più importanza, anche se lentamente, nelle scuole di tutto il mondo. Fu dalla metà degli anni 2000, tuttavia, il periodo in cui lo spostamento dell'attenzione verso la didattica delle materie scientifiche come scienze, tecnologia, ingegneria e matematica portò alla ribalta l'informatica come materia di studio scolastica. Oggi vi è un grande interesse internazionale nel rendere la didattica dell'informatica accessibile a tutti. Tale necessità porta alla luce l'importanza della ricerca sulla didattica dell'informatica. Gli studi di Papert riguardavano principalmente i bambini. La maggior parte degli studi sull'educazione informatica degli anni passati sono stati intrapresi con studenti appartenenti all'istruzione universitaria, i quali erano a loro volta un sottoinsieme privilegiato di studenti. Oggi la ricerca può fare grandi passi in avanti, anche grazie al fatto che l'istruzione è accessibile a molte più persone, ed è molto più semplice raccogliere dati.
Risulta dunque evidente come il campo della ricerca sulla didattica dell'informatica stia prendendo sempre più valore nel mondo. Sarà sempre più importante, dunque, definire nuovi approcci di didattica per l'informatica, creare nuovi strumenti e applicare nuovi metodi di ricerca cercando di comprendere cosa succede quando gli esseri umani imparano a controllare le macchine e come migliorare tali processi.

1.3 Teorie dell'apprendimento

L'apprendimento è un processo mediante il quale si acquisiscono nuove conoscenze. Le teorie dell'apprendimento descrivono come le persone imparano. Il processo di apprendimento della conoscenza è determinato, per ogni individuo, dall'intreccio fra componenti intuitive, quantitative e qualitative, sotto l'influenza di condizionamenti sociali, culturali ed emotivi. Questo è uno dei motivi per il quale questo argomento viene affrontato in svariate discipline come la psicologia, la pedagogia e la filosofia che hanno nel tempo prodotto una moltitudine di teorie. L'apprendimento è infatti un complesso di elementi dinamici, che segue spesso tracciati non lineari e che si può studiare in modo efficace solo mediante un approccio multidisciplinare. Quello dell'apprendimento è tuttavia un argomento ancora molto dibattuto in cui ci sono visioni e posizioni diverse che abbracciano diverse discipline. Data la moltitudine di teorie, per cercare di capire quali sono i tratti che più li accomunano e li contraddistinguono si parla di paradigmi di apprendimento o paradigmi educativi. Alcuni autori come [6] individuano alcuni paradigmi, in particolare:

- Comportamentismo
- Cognitivismo
- Costruttivismo

In questa sezione si vogliono presentare alcuni dei fondamenti teorici che hanno avuto una grande importanza per le scienze dell'apprendimento. Naturalmente, queste teorie non esistono isolatamente in quello che è il complesso processo dell'apprendimento ma, al contrario, queste teorie spesso interagiscono tra loro. Nei paragrafi seguenti verranno analizzate nello specifico le teorie del comportamentismo, cognitivismo e costruttivismo.

1.3.1 Comportamentismo

Alla base delle teorie comportamentiste c'è l'idea secondo cui sia possibile modificare il comportamento manifesto delle persone fornendo stimoli esterni. Tali stimoli sono in grado di fornire le risposte desiderate. Secondo queste dinamiche dunque l'apprendimento è inteso come modifica del comportamento osservabile negli studenti. Gli studenti

sono visti come dei contenitori vuoti, che ricevono passivamente le informazioni dal docente. L'interesse di studio in queste teorie è dunque incentrato sulla trasmissione della conoscenza intesa come capacità dello studente di rispondere agli stimoli esterni al fine di apprendere nuovi concetti. L'istruzione assume alcune caratteristiche tipiche di queste metodologie come l'apprendimento inteso come un processo ripetitivo. L'insegnante assume il ruolo di persona il cui scopo è quello di modellare il comportamento dello studente attraverso la stimolazione e la trasmissione dei concetti. Tra le diverse teorie comportamentiste è opportuno ricordarne due:

- Istruzione programmata: teoria che si basa sull'organizzazione dei contenuti mediante sequenze di comportamenti al fine di migliorare l'apprendimento. Questa teoria fu elaborata da Burrhus F. Skinner e si basa sulla concezione che le conoscenze progrediscono se il soggetto produce dei comportamenti desiderati che vengono rafforzati dall'effetto positivo prodotto. Il processo si divide in sequenze, si parte dal suddividere la conoscenza in piccole unità procedendo step dopo step. Successivamente gli studenti sono incentivati a fornire risposte ai problemi, mediante quiz ad esempio. Attraverso questa metodologia gli studenti apprendono in modo graduale e sequenziale iniziando con la soluzione di un problema semplice e proseguendo nella risoluzione di problemi sempre più complessi. Agli studenti viene poi restituito un feedback sulla base delle loro risposte, in questo processo dunque il feedback ha un ruolo fondamentale perché rappresenta la ricompensa a promuovere la nuova risoluzione di un problema e lo sviluppo dell'apprendimento.

- Mastery learning: teoria che consiste nell'articolazione di percorsi formativi suddivisi in blocchi, ciascuno caratterizzato da obiettivi di apprendimento da raggiungere da parte dello studente. Per passare al blocco successivo lo studente deve dimostrare di avere acquisito le conoscenze del blocco o dell'unità. Nonostante fu teorizzata intorno agli anni '70 dallo psicologo Benjamin Bloom rappresenta ancora oggi, nelle sue linee essenziali, un modello di riferimento per la didattica. La verifica dell'apprendimento è finalizzata al riscontro e all'accertamento dell'acquisizione dei prerequisiti di apprendimento e al loro eventuale consolidamento, con verifiche mirate ad individuare tempestivamente eventuali difficoltà.

1.3.2 Cognitivismo

Il cognitivismo condivide con le teorie comportamentistiche l'idea secondo cui lo studio dell'apprendimento deve essere oggettivo. Tuttavia, al contrario del comportamentismo le teorie cognitive ritengono di poter trarre informazioni sulla natura dei processi cognitivi studiando lo stato mentale delle persone. L'apprendimento delle informazioni avviene quando lo studente elabora le informazioni ed è inteso come un cambiamento nelle costruzioni mentali. L'idea comportamentista di osservare solo i comportamenti esterni viene superata dallo studio degli stati mentali interni degli individui. Il cervello umano è inteso come una vera e propria "macchina" capace di elaborare le informazioni. L'istruttore è il responsabile del processo di "input" delle informazioni, ma l'allievo è parte attiva nell'elaborazione delle informazioni e nel progettare e nel compiere l'apprendimento. Al cognitivismo appartengono molte importanti teorie come il modello dei "magazzini di memoria", l'idea del "carico cognitivo", le pre-concezioni e le misconcezioni. Argomenti che verranno discussi nello specifico nei capitoli successivi.

1.3.3 Costruttivismo

Dalla corrente cognitivista nacque una "seconda generazione" di cognitivismo che va sotto il nome di costruttivismo, che affonda le sue radici nelle opere di studiosi come Dewey, Vygotsky, Piaget. Il costruttivismo determina il passaggio da un approccio oggettivistico, ad uno soggettivistico. L'idea del costruttivismo è che la conoscenza non sia un elemento oggettivo caratteristico dell'essere umano ma è parte del sapere personale, frutto delle proprie esperienze. L'apprendimento è un processo di costruzione nella quale va enfatizzata la scoperta attiva. La costruzione di nuova conoscenza e quindi l'apprendimento diventa più efficace e padroneggiato non solo quando questo è frutto di un processo mentale ma anche quando è fortemente supportato da una costruzione reale come un'attività o un progetto.
Tra le più importanti teorie costruttiviste vi è il *costruzionismo*. Il costruzionismo condivide l'idea di base del costruttivismo secondo cui lo studente debba essere parte attiva e costruttore del suo sapere. Uno dei principali esponenti di questo movimento, specialmente per gli apporti forniti alla didattica e alle tecnologie dell'istruzione e dell'ap-

prendimento, è Seymour Papert. Papert [7] aggiunge al costruttivismo l'idea secondo cui siano necessari "oggetti con cui pensare", i quali sono i mattoni della conoscenza personale. Per il costruzionista nella scelta di questi materiali, però, non c'è solo un aspetto cognitivo, ma c'è anche una componente emotiva fondamentale. Papert, inoltre, privilegia la creazione di ambienti per l'apprendimento, come il linguaggio LOGO [5], un linguaggio e un ambiente di programmazione appositamente sviluppato per i bambini già discusso in precedenza. In questi ambienti è dunque, dove Papert ritiene che gli alunni siano in grado di esprimere la capacità di apprendimento. Il docente ha il compito di facilitare la scoperta e la costruzione del sapere e della conoscenza senza una impostazione tipicamente di visione istruttivista di pura trasmissione. Questa visione si discosta dalla visione dell'imparare per usare, ma si avvicina ad una visione del fare per imparare.

1.4 Le scienze cognitive e il ruolo dell'apprendimento e della memoria nella programmazione

Le scienze cognitive rappresentano l'insieme di discipline che hanno come oggetto di studio scientifico e filosofico la cognizione. Il fine comune di queste scienze è quello di comprendere il funzionamento della mente e dei suoi processi. Il campo è emerso negli anni '50, guidato in gran parte dal progresso della psicologia cognitiva. Le discipline solitamente incluse nel suo ambito sono filosofia, neuroscienze, psicologia, antropologia, linguistica e intelligenza artificiale. Le scienze cognitive negli anni hanno fortemente influenzato la didattica. Studiare come il cervello umano acquisisce, elabora informazioni e risolve i problemi è estremamente importante se si vuole sperimentare nuovi approcci didattici che siano in grado di migliorare sempre di più il processo di apprendimento nella didattica.

I ricordi nell'essere umano sono un aspetto fondamentale per il processo di apprendimento e di comprensione. Secondo le scienze cognitive un ricordo è un elemento di informazione che siamo in grado di richiamare alla mente. La parola memoria nelle scienze cognitive assume un significato generale che raggruppa una varietà di meccanismi cognitivi come l'elaborazione, l'immagazzinamento e il recupero delle informazioni. Atkinson e Shiffrin [8] sostengono che la memoria sia suddivisa in tre sistemi tra loro collegati che prendono

il nome di *registro sensoriale* (oggi prende il nome di memoria sensoriale), una *memoria a breve termine* anche definita con l'abbreviazione MBT e una *memoria a lungo termine* o MLT. Di seguito viene descritta per ognuna di questa, una breve introduzione utile a comprendere concetti dei capitoli successivi.

- **Memoria sensoriale**: si riferisce alla caratteristiche fisiche che lo stimolo ha suscitato nella mente. Costituisce la prima fase del processo di immagazzinamento delle informazioni. Di solito è studiato nella modalità visiva (memoria iconica), ma include anche l'udito (memoria sensoriale uditiva) e il tatto (memoria tattile). Negli studi e negli esperimenti effettuati sugli esseri umani per questo tipo di memoria, dopo un breve periodo di esposizioni a stimoli esterni (anche meno di un secondo) questi sono in grado di richiamare alcuni dettagli. In questa fase, dunque, le informazioni restano nella memoria per un brevissimo tempo, sono gli organi di senso a selezionare e giudicare quelle maggiormente importanti.

- **Memoria a breve termine**: questo tipo di memoria è in grado di mantenere attive le informazioni per un tempo necessario a svolgere compiti cognitivi. La informazioni vengono suddivise in piccole quantità anche denominate *chunk*, è tema di dibattito quanti elementi la memoria a breve termina sia in grado di memorizzare, generalmente gli studiosi concordano per un numero pari a 4 ± 1 elementi o 7 ± 2 elementi. La memoria a breve termine è caratterizzata da una capacità estremamente limitata. La memoria a breve termine è un tema di grande interesse per la didattica dell'informatica, comprendere come funziona può essere importante per suggerire nuovi metodi di apprendimento. Nello specifico la programmazione è un tipo di task che implica la necessità di memorizzare informazioni relative allo stato dei processi in esecuzione, alla progettazione, ai valori della variabili e agli obiettivi generali del programma. Rilevante in questo contesto è anche il concetto di carico cognitivo, che verrà discusso e approfondito nei capitoli successivi.

- **Memoria a lungo termine**: al contrario della memoria a breve termine la memoria a lungo termine è in grado di immagazzinare informazioni per un tempo indeterminato. La codifica di un informazione, nel caso della memoria a lungo termine, diventa un processo attraverso il quale il cervello altera la sua struttura

interna in modo da creare una rappresentazione duratura nel tempo. In questo processo sono diverse le regioni del cervello coinvolte, ma la creazione di diverse rappresentazioni di informazioni sembrano dipendere dal meccanismo comune delle connessioni sinaptiche tra i neuroni. Sono svariate le dinamiche che possono determinare il successo o il fallimento della codifica delle informazioni in questo tipo di memoria. Questi includono fattori esterni, come distrazioni o stress, oppure cognitivi. Nella didattica dell'informatica studiare come funziona questa memoria è importante per determinare alcuni meccanismi tipici della programmazione in particolare per chi si approccia per la prima volta a questa disciplina. Alcuni fenomeni, infatti, sono tipici dei programmatori alle prime armi come la prevalenza di una conoscenza di base molto "fragile" che può mancare in questo tipo di memoria o anche essere dimenticata, oppure conoscenza appresa ma mai utilizzata o conoscenza acquisita in un contesto completamente differente da quello in cui si opera. Di questi problemi verrà ampiamente discusso nel capitolo successivo che sarà, in particolare, incentrato sui processi cognitivi e sull'apprendimento riguardanti la programmazione.

Capitolo 2

Imparare ed insegnare a programmare

Tra i principali argomenti di studio e di ricerca della didattica dell'informatica c'è la programmazione. In particolare, si cerca di comprendere come uno studente alle prime armi impari a programmare e quali sono le principali difficoltà che trova durante il suo percorso di apprendimento. L'introduzione alla programmazione pone grandi sfide per gli studenti. Programmare non è facile. I corsi di introduzione alla programmazione spesso hanno un alto tasso di fallimento e scarsi risultati per gli studenti. La programmazione dunque pone sfide non solo allo studente ma anche al docente. Individuare come gli studenti comprendono le dinamiche della programmazione e come applicano i concetti teorici è un argomento di grande interesse per i docenti di informatica, i quali naturalmente desiderano che i loro studenti imparino con successo e con ottimi risultati questa disciplina. Programmare è complesso in quanto richiede un grande sforzo cognitivo e non solo. I linguaggi di programmazione sono costrutti artificiali complessi: come le regole grammaticali del linguaggio naturale, sono costituite da un numero relativamente piccolo di elementi che possono essere combinati in infiniti modi. Il campo della didattica dell'informatica e della programmazione in particolare ha suscitato notevole interesse anche per la comunità scientifica. In letteratura esistono molti lavori, che verranno analizzati, ma molte questioni molto importanti rimangono ancora aperte. Vi è, dunque, la necessità oggi di individuare nuove forme di didattica della programmazione al fine di

agevolare quello che è il processo dell'apprendimento.
Nel seguente capitolo verranno trattati i principali argomenti di studio e di ricerca che vertono sulla didattica della programmazione, ponendo grande attenzione alle principali difficoltà cognitive che gli studenti novizi hanno quando imparano a programmare. La maggior parte degli studi scientifici analizzati in questo capitolo hanno come soggetto di studio studenti che frequentano corsi di programmazione di base (Principles of Computer Science, spesso abbreviato in "CS1").

2.1 Imparare a programmare è difficile

Coerentemente con i problemi storici che circondano la programmazione discussi nel capitolo 1, la didattica della programmazione è sempre stata tipicamente considerata difficile per gli studenti. I corsi di introduzione alla programmazione spesso hanno un alto tasso di fallimento e scarsi risultati per gli studenti. Tuttavia, non esistono statistiche mondiali sui tassi di fallimento, sui tassi di abbandono o sui tassi di superamento dei corsi di introduzione alla programmazione a livello universitario. Bennedsen e Caspersen in [9] hanno cercato di quantificare questo problema esaminando 63 istituti di istruzione internazionali (il 63% provenienti dagli USA), al fine di individuare il numero di studenti che abbandonano, non si presentano all'esame o non passano l'esame. I dati raccolti, relativi a studenti di corsi sulla introduzione alla programmazione (CS1), hanno evidenziato un tasso di fallimento in media di circa il 33%, ma con enormi divari tra istituti scolastici, dato principalmente dalla diversità di paesi ed altri aspetti come condizioni economiche, sociali e politiche. In modo analogo anche Watson e Li in [10] hanno analizzato dati sul tasso di fallimento degli studenti per i corsi introduttivi di programmazione ed hanno raccolto conclusioni molto simili al precedente studio. Sono state estratte e analizzate le percentuali di superamento che descrivono i risultati di 161 corsi svolti in 15 paesi diversi, in 51 istituti. È stato riscontrato un tasso medio di superamento mondiale del 67,7%. Gli autori hanno evidenziato, inoltre, che le percentuali di superamento non differivano significativamente nel tempo o in base al linguaggio di programmazione insegnato durante il corso. Entrambi gli studi analizzati concludono che, nonostante i dati e le percentuali emerse dalle analisi sui corsi di programmazione possano sembrare alte, non sono partico-

larmente allarmanti. Tuttavia entrambi gli studi non hanno effettuato un confronto tra il corso di programmazione ed altri corsi. Luxton-Reilly e Peterson [11] hanno confrontato la percentuale di superamento dell'esame rilevato dagli studi precedenti di circa il 67%, con la percentuale di superamento di esami relativi ad altri corsi in Nuova Zelanda che risulta essere dell'82%, più alta del 15Altri studi dimostrano che molti studenti, anche dopo aver passato con successo l'esame di programmazione, mantengono solo una comprensione basilare e non approfondita della programmazione. Un importante lavoro che dimostra quanto detto è stato fatto da McCracken et al. [12]. Il gruppo di McCracken ha analizzato le competenze relative alla programmazione che gli studenti hanno acquisito al completamento del corso di introduzione alla programmazione. Il gruppo ha valutato 216 studenti utilizzando un insieme di problemi di programmazione selezionati in modo tale da essere risolto da qualsiasi studente che abbia frequentato un corso di introduzione alla programmazione. Dall'analisi dei test è risultato che la maggior parte degli studenti ha ottenuto risultati peggiori di quanto si aspettassero i loro insegnanti, e la maggior parte non è riuscita a portare a termine i problemi assegnati. Analizzando i dati delle università di diversi paesi, il gruppo ha evidenziato che i principali problemi osservati sulle capacità di programmazione sono indipendenti dal paese e dal sistema educativo. Il problema comune a tutti gli studenti è il processo di astrazione del problema da risolvere data la sua descrizione. Hanno osservato, inoltre, che nella maggior parte delle università, la principale lamentela degli studenti era la mancanza di tempo per completare gli esercizi assegnati. Data l'importanza e la natura internazionale di questo studio, i risultati descritti sono ampiamente considerati significativi e convincenti.
Gli studi descritti evidenziano dunque la difficoltà nell'apprendere la programmazione, anche per chi è stato in grado di passare l'esame. Complesso è anche insegnare in modo efficace le abilità della programmazione. I risultati dello studio hanno mostrato che la maggior parte degli studenti partecipanti non è riuscita a raggiungere uno degli obiettivi fondamentali di un corso di informatica del primo anno: acquisire almeno un livello di base di abilità con la programmazione.

2.1.1 Si può essere portati per la programmazione?

Se molti studi concordano sul fatto che programmare sia difficile per gli studenti novizi e i risultati agli esami di programmazione hanno un alto tasso di fallimento, altri studi affermano che, sebbene ci sia un grande numero di studenti con scarsi risultati, vi sia una parte di studenti che, al contrario, mostrano capacità ben oltre la media ottenendo i massimi risultati. Sulla base delle analisi viste fino ad ora pare che la programmazione sia "semplicemente" un'attività difficile da imparare rispetto ad altre materie scolastiche. Tuttavia, è interessante notare come i tipici corsi di programmazione di base abbiano un consistente gruppo di studenti con un insolito tasso di eccellenti risultati: sembra che ci sia un sottoinsieme significativo di studenti che sia in grado di imparare la programmazione in modo facile ed intuitivo, ottenendo risultati oltre la media. Dehnadi e Bornat [13] sostengono che tra gli studenti sia presente un fenomeno di "doppia gobba" che consiste in una netta suddivisione tra studenti in grado di ottenere ottimi risultati, ed altri con scarsi risultati. Gli autori hanno sperimentato un test nel quale dimostrano di prevedere il successo o il fallimento anche prima che gli studenti abbiano avuto contatti con qualsiasi linguaggio di programmazione con altissima precisione. Il test sperimentato da Dehnadi e Bornat consisteva nell'individuare tre principali ostacoli che fanno inciampare gli studenti principianti. Nell'ordine:

- Assegnamento e sequenza
- Ricorsione e iterazione
- Concorrenza

Nonostante i risultati dei test mostrino tale fenomeno, non ci sono rilevanti studi scientifici in grado di dimostrare che alcune persone possiedono una naturale attitudine alla programmazione, ed inoltre, non è noto se la capacità di programmazione sia correlata all'età, al sesso o al livello di istruzione. Né è stata trovata alcuna correlazione ad attitudini misurate nei test convenzionali di "intelligenza" o di "capacità di risoluzione dei problemi".

2.1.2 Bipartizione dei voti

La questione della netta bipartizione nei voti tra studenti è stata oggetto di notevole attenzione da parte della letteratura nell'ambito della didattica della programmazione. Il termine *bimodale* viene spesso utilizzato in questi casi per descrivere le distribuzioni bimodali di voti, con tassi più alti del solito sia di fallimento che di voti alti. Il fenomeno della bimodalità nei voti tuttavia, non è condiviso da tutti e viene spesso criticato. Robins [14], infatti, propone una spiegazione alternativa alla distribuzione bimodale dei voti di un tipico corso di programmazione di base. Robins ipotizza l'esistenza dell'effetto *Learning Edge Momentum* (LEM). Il learning edge momentum si basa sul principio, ben accettato nella letteratura psicologica ed educativa, che apprendiamo ai "margini" di ciò che sappiamo. L'affermazione centrale dell'ipotesi Learning Edge Momentum è che, dato un dominio di nuovi concetti da apprendere, qualsiasi apprendimento avvenuto con successo rende in qualche modo più facile acquisire ulteriori concetti correlati al dominio a cui si fa riferimento. Al contrario, l'apprendimento che ha avuto insuccesso, rende tale dominio più difficile, e produrrà inevitabilmente insuccesso nell'apprendimento. A seconda della forza di questo effetto, il successo precoce o il fallimento nell'acquisizione di concetti può rafforzarsi, creando uno slancio verso risultati bimodali. L'effetto LEM, secondo Robins, varia a seconda delle proprietà del dominio target, in particolare dipende dalla misura in cui i concetti nel dominio sono indipendenti o integrati. Robins, inoltre, esclude l'assunto secondo cui ci siano due diverse popolazioni di persone, quelle che possono imparare a programmare e quelle che non possono. Come evidenziato nel paragrafo precedente, infatti, nonostante decenni di ricerche approfondite, non è stato possibile identificare il fattore o i fattori che caratterizzano questi due ipotetici gruppi. Robins sostiene anche che i test attitudinali non sono predittori affidabili e in passato non hanno identificato alcuna capacità cognitiva particolarmente significativa per l'apprendimento della programmazione. Nonostante i dibattiti sulla terminologia, è importante tenere conto dei risultati mostrati fino ad ora. Per una università, se un determinato corso ha un tasso di fallimento più elevato rispetto a tutti gli altri corsi (della stessa università) è un motivo di preoccupazione. Nel caso dei corsi di programmazione, sebbene vi siano alte percentuali di fallimento ci sono, d'altro canto, alte percentuali di risultati eccellenti. Se, dunque, da un lato vi sono molti dibattiti (tuttora vivi) sul termine bimodale, vi sono

studi che dimostrano la totale inesistenza di voti bimodali nei corsi di introduzione alla programmazione. Patitsas et al. [15] hanno analizzato statisticamente 778 distribuzioni di voti dei corsi di programmazione introduttiva di una grande università e hanno riscontrato che solo il 5,8% delle distribuzioni ha dimostrato di possedere una multimodalità nei voti. Gli autori dello studio hanno inoltre ideato un esperimento di psicologia per capire perché i docenti di informatica ritengono che i loro voti siano bimodali. Dall'esperimento psicologico hanno scoperto che spingere i partecipanti a pensare alla percezione comune dei risultati bimodali porta ad una maggiore probabilità dei partecipanti di etichettare le distribuzioni ambigue come bimodali. Ciò indica che esiste un forte bias cognitivo di conferma, che gioca un ruolo fondamentale nella convinzione che i risultati bimodali siano tipici nei corsi di programmazione introduttiva. Dall'analisi psicologica Patitsas et al., inoltre, affermano che la percezione dei voti bimodali nell'informatica è una difesa sociale del comparto docenti. Gli autori infatti sostengono che sia più facile per i docenti credere che alcuni studenti siano portati alla programmazione e altri no, piuttosto che fare i conti con le carenze degli approcci pedagogici e degli strumenti di valutazione.

2.2 La programmazione per studenti alle prime armi

I corsi introduttivi sulla programmazione sono sempre stati, tipicamente, considerati difficili per gli studenti. Come discusso nel paragrafo precedente, una gran parte di loro ottiene risultati non soddisfacenti e altri superano l'esame con grande difficoltà. L'interesse dei docenti che insegnano corsi introduttivi alla programmazione è quello di fornire un contesto di apprendimento che sia efficace ed utile all'apprendimento della programmazione per studenti che, molto probabilmente, non hanno mai avuto a che fare con linguaggi di programmazione e concetti teorici connessi ad essa. I programmatori esperti sono bravi a riconoscere, utilizzare e adattare modelli e schemi nella programmazione. Sono più veloci e più precisi e sono in grado di attingere a un'ampia gamma di esempi, fonti di conoscenza e strategie efficaci (caratteristiche che contraddistinguono gli esperti dai programmatori novizi). Per definizione, infatti, i novizi non hanno tutti questi punti di forza. Gli studi esaminati da Winslow [16], ad esempio, hanno concluso che i principianti sono limitati alla conoscenza superficiale e organizzata in modo inconsistente.

Nelle menti degli studenti novizi mancano i modelli mentali (concetto che sarà trattato nei paragrafi successivi) dettagliati e non riescono ad applicare la conoscenza pertinente al dominio e si avvicinano alla programmazione "riga per riga" piuttosto che utilizzare "blocchi" o strutture ben definite nella programmazione. Winslow sostiene infatti che sia generalmente accettato che ci vogliano circa 10 anni per trasformare un principiante in un programmatore esperto. Perkins et al. [17] hanno distinto tre principali categorie di studenti novizi :

- Stoppers: studenti che tendono a fermarsi e ad arrendersi quando non riescono a trovare immediatamente una soluzione o un'intuizione su come procedere con un problema.
- Movers: studenti che provano ad utilizzare diversi approcci per risolvere un problema.
- Tinkerers: studenti che cercano di risolvere un problema di programmazione scrivendo del codice e quindi apportando piccole modifiche nella speranza di farlo funzionare, con poche possibilità di progresso.

Lhatinen et al. [18] hanno analizzato i risultati di alcuni questionari effettuati a circa 550 studenti di un corso base di programmazione e 34 professori. I risultati dell'indagine hanno evidenziato che i concetti di programmazione più difficili da apprendere sono quelli che richiedono la comprensione di entità più ampie ed astratte del programma piuttosto che dei singoli concetti teorici. Tra i concetti con cui gli studenti hanno avuto maggiori difficoltà, ad esempio, vi sono i puntatori e la gestione della memoria. I risultati hanno inoltre mostrato che argomenti come Input e Output o le librerie dovrebbero ricevere maggiore attenzione, in quanto questi argomenti non erano inclusi nelle spiegazioni durante il corso. Dalla ricerca è inoltre emerso che, sia gli studenti che i docenti, hanno convenuto che il miglior modo per apprendere i concetti teorici era attraverso la pratica e le esercitazioni di programmazione. Gli studenti, in particolare i novizi, necessitano di mettere subito in pratica i concetti appresi per essere compresi in maniere più efficace. Quanto più pratiche e concrete sono le esercitazioni e i materiali didattici del corso, tanto più lo studente è in grado di apprendere i concetti di programmazione in maniera efficace. Il problema più grande dei programmatori alle prime armi dunque non

sembra essere la comprensione dei concetti di base, ma piuttosto imparare ad applicarli. Robins et al. [14], infatti, suggeriscono che gli insegnanti dovrebbero concentrarsi maggiormente su questi tipi di caratteristiche, specialmente sulle questioni che sono alla base della progettazione del programma. Inoltre, uno dei problemi nell'insegnamento della programmazione sembra anche essere quello che gli studenti sovrastimano la loro comprensione [18]. Secondo Lahtinen, infatti, gli insegnanti sono portati a pensare che i contenuti del corso di programmazione siano più difficili per gli studenti rispetto a quanto gli studenti stessi dicono. La ragione delle diverse percezioni di difficoltà può dipendere dal fatto che gli studenti non si rendono conto di tutte le difficoltà che l'apprendimento pone. Gli insegnanti invece essendo più esperti se ne rendono conto facilmente anche, ad esempio, quando valutano gli esami. Inoltre, gli insegnanti conoscono i concetti più a fondo e sono in grado di individuare facilmente se gli studenti non hanno costruito una piena comprensione degli argomenti. Gli studenti al contrario tendono a sottostimare la difficoltà dei concetti e a credere di aver compreso concetti che in realtà non sono stati compresi perfettamente o sono stati compresi in parte in modo errato.

2.3 Le principali difficoltà cognitive nella programmazione per studenti novizi

Come anticipato nel capitolo 1, nella storia, sono stati tanti gli studi effettuati per cercare di capire come gli studenti imparano a programmare. In particolare si è cercato di attingere dalle scienze cognitive. Le scienze cognitive, infatti, negli anni che hanno fortemente influenzato, nell'ambito della didattica, le metodologie e gli approcci di insegnamento. Durante gli anni '70 fino agli anni '90, c'è stata una grande attenzione sulla psicologia della programmazione. Il ruolo dell'apprendimento e della memoria nella programmazione è stato ampiamente trattato nella letteratura includendo concetti della psicologia cognitiva, come la rappresentazione della conoscenza, la risoluzione dei problemi e la memoria di lavoro. Quando si ha a che fare con la programmazione è importante studiare come la mente umana è in grado di elaborare le informazioni e risolvere i problemi mediante la scrittura di istruzioni in forma di codice.
Nel capitolo precedente si è discusso della suddivisione della memoria in tre diversi ti-

pi (memoria sensoriale, memoria a breve termine e memoria a lungo termine) e cosa implicano nel processo di apprendimento della programmazione. In questa sezione si discuterà, invece, nello specifico delle principali difficoltà cognitive che gli studenti novizi possono avere quando imparano a programmare.

2.3.1 I concetti soglia

Nell'ambito della didattica dell'informatica una teoria che ha ricevuto grande attenzione è stata quella dei *concetti soglia*. I concetti soglia sono quei concetti che rappresentano le sfide chiave nell'apprendimento di un determinato dominio di conoscenza; soglie oltre le quali, se acquisiti tali concetti con successo, consentono una migliore comprensione dell'argomento. Meyer e Land [19] hanno identificato cinque caratteristiche dei concetti soglia:

1. I concetti soglia sono caratteristici di ciascuna disciplina. Ciò significa che alcuni concetti soglia sono particolari e distintivi in un contesto specifico. Gli stessi concetti non possono essere applicati per più discipline differenti. In questo modo i concetti soglia aiutano anche a definire l'ambito del dominio della conoscenza.

2. I concetti soglia implicano forme di "conoscenza fastidiosa" o nozioni che appaiono illogiche, non familiari o differenti da ciò che ci si aspettava. Per alcuni concetti soglia sorgono problemi essendo difficili da comprendere o da integrare nel processo di apprendimento. Possono risultare "fastidiosi" a causa del modo in cui i concetti fondamentali sono legati insieme in una dinamica sottostante che può essere impercettibile per gli studenti alle prime armi.

3. I concetti soglia sono integrativi, ovvero permettono di costruire nuove connessioni e relazioni varcando la soglia. Oltrepassate queste soglie, infatti, è possibile creare nuove connessioni e modelli mentali relativi ad un determinato argomento di dominio.

4. La trasformazione nella comprensione associata al superamento dei concetti soglia di solito non è una dinamica reversibile. Una volta che sono state individuate nuove connessioni e modelli, non è possibile tornare facilmente a modelli di comprensione

precedenti. Tuttavia, potrebbe accadere che il concetto stesso potrebbe essere sostituito da concettualizzazioni alternative ancora più sofisticate.

5. Il coinvolgimento dato dal successo del superamento dei concetti soglia è trasformativo. Superati tali concetti nella mente dello studente si creeranno nuovi modi di vedere, comprendere e descrivere i concetti. La rappresentazione dei modelli di comprensione, che risulta dall'impegno dello studente, essendo (come detto nel punto quattro) irreversibile, avrà effetti nella mente andando a cambiare quelli che sono i modelli mentali (concetto di cui si discuterà nei paragrafi successivi). Il "mondo" viene dunque rivisto, cambiando anche il modo in cui si pensa e si agisce.

I concetti soglia nella programmazione

I concetti soglia hanno attirato una notevole attenzione nella didattica dell'informatica e della programmazione. Shinners-Kennedy e Finche [20] hanno approfondito la letteratura riguardo i concetti soglia cercando di individuarli nel contesto dell'apprendimento della programmazione. Gli autori hanno analizzato uno studio internazionale effettuato nel 2005 da un gruppo di ricercatori nel campo dell'educazione per identificare i concetti di soglia nell'informatica. I membri del gruppo provenivano da diversi paesi dell'Europa e degli Stati Uniti e hanno utilizzato una varietà di strumenti e strategie nel tentativo di fornire prove empiriche a sostegno dell'identificazione di uno o più concetti soglia nella programmazione. Questo studio rappresenta uno dei tentativi più significativi e variegati sull'individuazione dei concetti soglia nella disciplina dell'informatica e della programmazione. Un'indagine iniziale informale fu effettuata su 36 istruttori di 9 paesi diversi ed ha identificato 33 concetti tipici dell'informatica. Tra i quali i più comuni sono:

- Livelli di astrazione
- Puntatori
- Distinzione tra classi, oggetti e istanze
- Ricorsione

- Astrazione procedurale
- Polimorfismo

Shinners-Kennedy e Finche, tuttavia, ritengono che, nonostante il grande sforzo effettuato dal gruppo di ricerca i ricercatori, nella difficoltosa indagine dei concetti soglia nella programmazione, siano arrivati ad un "vicolo cieco". Gli autori concludono che probabilmente la difficoltà nel definire e trovare i concetti soglia nella programmazione sia dovuta al fatto che si tenta di considerare i limiti intrinseci della mente umana considerando i bias e la sfera emotiva, che con molta probabilità forniscono prove inaffidabili.

2.3.2 I modelli mentali

Un modello mentale è un modello, appartenente alla mente umana, di come funzionano alcuni aspetti del mondo che ci circonda. Il termine modello mentale fu introdotto nell'ambito delle scienze cognitive all'inizio degli anni '80 da Johnson-Laird [21], il quale definì i modelli mentali come le strutture di base della cognizione umana. In un'altra importante pubblicazione Gentner e Stevens [22] hanno introdotto il concetto di modello mentale nell'ambito dell'interazione tra uomo e tutto ciò che lo circonda. Secondo gli autori quando l'essere umano interagisce con l'ambiente, con la tecnologia o con altri umani, è in grado di sviluppare modelli mentali. I modelli mentali sono dunque dei meccanismi cognitivi appartenenti all'uomo che hanno delle ragioni evolutive ben delimitate: hanno un potere predittivo in quanto possono essere utilizzati per comprendere il comportamento osservato nel mondo e ragionare sui comportamenti futuri.
Le caratteristiche tipiche dei modelli mentali secondo la descrizione di Norman [23] sono:

- Riflettono le convinzioni delle persone sui sistemi che usano e sui propri limiti e rappresentano il grado di incertezza che le persone provano riguardo ai diversi aspetti della loro conoscenza;
- Forniscono spiegazioni talvolta errate e semplificate di fenomeni complessi;
- Possono contenere solo descrizioni incomplete e parziali delle operazioni e possono contenere enormi aree di incertezza;

- Sono non scientifici e imprecisi e spesso si basano su supposizioni e convinzioni ingenue, nonché su regole superstiziose che sembrano funzionare anche se non hanno alcun senso;
- Si evolvono nel tempo man mano che le persone interagiscono con i sistemi e modificano i loro modelli per ottenere risultati desiderati e realizzabili;
- Sono soggetti a modifiche in qualsiasi momento;
- Possono essere "eseguiti" per simulare e prevedere mentalmente il comportamento del sistema, sebbene la capacità delle persone di eseguire modelli sia limitata.

I modelli mentali nella programmazione

Nell'ambito della programmazione sono stati effettuati molti studi sui modelli mentali, al fine di comprendere quali siano gli effetti positivi o negativi nel processo di apprendimento degli studenti. Uno studente in grado di concretizzare adeguati modelli mentali sarà in grado di comprendere meglio i concetti della programmazione. A tal riguardo Ma et al. [24] hanno realizzato un'indagine sull'efficacia dei modelli mentali utilizzati dai programmatori alle prime armi alla fine di un corso di programmazione Java di base. L'analisi è stata effettuata mediante test a circa 120 studenti su concetti base della programmazione (in particolare sulla programmazione ad oggetti) come assegnamento di valore e riferimento all'oggetto (reference). Il test è stato progettato per individuare i modelli mentali. Per l'assegnazione di un valore ad una variabile è stato mostrato un breve pezzo di codice nel quale veniva copiato il valore di una variabile su un'altra variabile. Per la reference, un oggetto viene copiato e memorizzato in una variabile di tipo oggetto a sua volta, come segue:

```
Person a, b;
a = new Person ("Jack");
b = new Person ("Tom");
b = a;
```

Il questionario conteneva due parti: la parte delle domande a risposte aperte e la parte delle domande a risposte multiple. La domanda a risposta aperta chiedeva ai par-

tecipanti di descrivere l'esecuzione di un piccolo programma, come quello mostrato sopra, utilizzando testo o diagrammi. La scelta della domanda aperta e non strutturata è stata fatta in quanto gli autori si aspettavano che rivelasse informazioni sui modelli mentali dei partecipanti. Il questionario a scelta multipla conteneva domande che chiedevano ai partecipanti di prevedere il risultato dell'esecuzione di un piccolo programma da una raccolta di opzioni di risposta predefinite. I risultati hanno mostrato una raccolta di modelli mentali sui concetti di assegnazione e reference da parte dei programmatori inesperti rivelando che, al termine del primo anno di corso, un terzo degli studenti possedeva ancora modelli mentali errati sul concetto di assegnazione di valore, e solo il 17% degli studenti era in possesso di modelli mentali utili alla comprensione del concetto di reference. Sia l'assegnamento che la reference sono concetti chiave nella programmazione orientata agli oggetti. Se il dato che solo il 17% aveva creato i giusti modelli mentali per il concetto di reference (che di per sé può risultare molto complesso per uno studente novizio) suscita preoccupazione, il dato più allarmante è il fatto che per un concetto elementare della programmazione come quello dell'assegnamento il 33% circa degli studenti non sia stato in grado di acquisire i giusti modelli mentali neanche alla fine del corso. Non a caso, i risultati mostrano anche che gli studenti con modelli mentali adeguatamente corretti hanno ottenuto risultati significativamente migliori nell'esame del corso e nei compiti di programmazione rispetto a quelli con modelli mentali errati. Ciò rivela quanto sia importante che i programmatori inesperti sviluppino modelli mentali giusti per imparare a programmare e per comprendere i concetti chiave teorici.

Tuttavia, è importante evidenziare che i modelli mentali non sono tutti uguali. Data, quindi, l'assoluta diversità dei modelli mentali appartenenti ad ogni singolo studente è ragionevole precisare che sviluppare modelli mentali corretti, per i principali concetti teorici della programmazione, può anche richiedere del tempo. Può, inoltre, verificarsi che attuali modelli mentali conservati dagli studenti siano solo parzialmente funzionali e che, col tempo, possano migliorare e perfezionare la capacità di creare modelli corretti. Gestire questa diversità all'interno di un gruppo e aiutare gli studenti a sviluppare modelli mentali corretti è una vera sfida per gli insegnanti di informatica.

2.3.3 La macchina concettuale

Un tipo di modello mentale è quello che gli studiosi chiamano *macchina concettuale* ("notional machine"). Una macchina concettuale è un'astrazione della mente umana che spiega come i programmi vengono eseguiti in un determinato linguaggio o famiglia di linguaggi strettamente correlati. Una macchina concettuale è, dunque, un modello mentale dell'esecutore che lo studente si crea quando impara a programmare. Ci possono essere differenti macchine concettuali per un dato linguaggio, che riflettono obiettivi, gradi di sofisticazione e livelli di astrazione differenti. D'altro canto, differenti linguaggi di programmazione possono anche condividere alcune implementazioni tali da rendere le macchine concettuali molto simili. Molti linguaggi ad alto livello, infatti condividono alcune caratteristiche tali da renderli somiglianti dal punto di vista di una macchina concettuale. Tuttavia, in maniera analoga, esistono linguaggi che possiedono macchine concettuali significativamente diverse, in quanto intrinsecamente differenti. Una macchina concettuale di un linguaggio di programmazione, come Java orientata agli oggetti, può essere molto diversa da una macchina concettuale di un linguaggio funzionale come Lisp.
Sorva [25] sostiene che una macchina concettuale non è una rappresentazione mentale che uno studente ha del computer. Sono gli studenti che creano modelli mentali di macchine concettuali. Una macchina concettuale non è nemmeno una "descrizione" o una "visualizzazione" del computer, sebbene descrizioni e visualizzazioni di una macchina concettuale possano essere create dagli insegnanti per gli studenti. Sorva, inoltre, sintetizza il concetto di macchina concettuale in 6 punti:

- È un'astrazione idealizzata dell'hardware del computer e di altri aspetti dell'ambiente di runtime dei programmi;
- Ha lo scopo di far comprendere cosa succede durante l'esecuzione del programma;
- È associato a uno o più paradigmi o linguaggi di programmazione ed eventualmente a un particolare ambiente di programmazione;
- Consente di descrivere la semantica del codice di programma scritto in quei paradigmi e linguaggi (o sottoinsiemi di essi);

- Dà una prospettiva particolare riguardo l'esecuzione dei programmi;
- Riflette correttamente ciò che fanno i programmi quando vengono eseguiti.

La macchina concettuale, dunque, è un tipo di modello mentale che lo studente, in particolare quello alle prime armi con la programmazione, plasma in modo frequentemente errato, incompleto, non scientifico e soggetto a cambiamento in qualsiasi momento. Può essere basato su congetture che attingono a caratteristiche superficiali del programma. Tuttavia, nonostante tali carenze, lo studente può sentirsi a proprio agio con la macchina concettuale creatasi e può fare affidamento su di essa durante lo sviluppo di altri modelli mentali per la programmazione. I novizi possono, inoltre, utilizzare modelli multipli, spesso contraddittori, per affrontare le sfide della programmazione. Al contrario, i modelli mentali degli esperti sono più stabili e accurati e si basano su principi generali piuttosto che su caratteristiche superficiali. Una sfida della didattica della programmazione è facilitare la creazione di giuste macchine concettuali in modo che acquisiscono le caratteristiche dei modelli mentali di programmatori più più esperti. Secondo Sorva, infatti, aiutare la formazione di modelli mentali il prima possibile è importante, poiché sarà molto più difficile eliminare una macchina concettuale sbagliata e radicata nella mente degli studenti, piuttosto che costruirne insieme allo studente una nuova. La ricerca sui modelli mentali prevede inoltre che ci si può aspettare che i programmatori inesperti abbiano difficoltà a trasferire le loro macchine concettuali riguardo ad uno specifico linguaggio di programmazione verso altri linguaggi, anche se questi sono molto simili a quelli imparati, ammenochè il modello originale non sia stato ben compreso attraverso una notevole quantità di pratica e teoria.
Lo studente che frequenta un corso di base di informatica ha, dunque, bisogno di costruire un modello mentale di una macchina concettuale adeguata e corretta per poter imparare a programmare. Molti autori hanno discusso del ruolo della macchina concettuale nell'introduzione alla programmazione in termini di modelli mentali. Alcuni attribuiscono la fragile conoscenza e comprensione della programmazione degli studenti, in gran parte, alla mancanza di una corretta macchina concettuale. Un modello mentale di una macchina concettuale corretta, infatti, insieme alla comprensione del programma consente ad un programmatore di fare inferenze sul comportamento del programma e di prevedere modifiche future ai programmi che sta scrivendo. Un principiante avrà solo un modello

mentale del sistema specifico che sta utilizzando per la programmazione, e di volta in volta che acquisisce esperienza, forma modelli mentali di altre macchine concettuali e schemi sempre più generali e validi della programmazione.

Perché è importante comprendere le macchine concettuali

Comprendere come macchine concettuali diverse per lo stesso linguaggio influiscono sulla formazione di modelli mentali degli studenti che si approcciano per la prima volta alla programmazione è un problema aperto ed estremamente significativo. Individuare se alcune macchine concettuali si dimostrano più efficaci e valide da imparare per gli studenti alle prime armi può essere oggetto di studio per approfondimenti più ampi sulla didattica della programmazione. Questi studi potrebbero avere anche delle notevoli implicazioni per quanto riguarda l'ordine con cui si introducono i concetti teorici o le caratteristiche strutturali di un linguaggio di programmazione. Diverse macchine concettuali, infatti, possono richiedere un grande carico cognitivo (concetto che verrà approfondito nei paragrafi successivi) che può inficiare negativamente sull'apprendimento della programmazione di base. Ad esempio, potrebbe accadere che la programmazione imperativa sia molto più facile per scrivere programmi velocemente (poiché per lo studente è più semplice individuare collegamenti ordinati tra le parti di un programma), ma questi stessi benefici potrebbero rendere l'apprendimento di concetti teorici più complessi e il debug molto più difficile. La programmazione funzionale, al contrario, potrebbe avere le problematiche opposte. Per questo motivo è importante effettuare studi sulle macchine concettuali. Individuare le macchine concettuali che più comunemente gli studenti si formano, renderebbe la didattica dell'informatica molto più efficace sotto ogni aspetto. In letteratura, tuttavia, è molto raro trovare studi nel quale si cerca di indagare eventuali macchine concettuali anche dopo alcuni mesi dal corso di programmazione di base, è dunque evidente che c'è ancora molto lavoro da fare. La comprensione di questi problemi lungo queste differenti dimensioni dovrebbe darci una comprensione molto più incisiva e sofisticata dell'impatto di paradigmi, stili di programmazione e metodologie didattiche rispetto a quella che troviamo attualmente in letteratura.

Macchina concettuale e il tracing nella programmazione

2.3.4 Le misconcezioni

Un altro modello mentale sul quale la letteratura si è spesso concentrata è quello delle *misconcezioni*. Una misconcezione è un "idea errata" che lo studente può generare durante il processo di apprendimento di un determinato concetto. In alcune discipline, concetti e princìpi sono spesso soggetti a differenti interpretazioni e rappresentazioni. Gli studenti, spesso, sono incoraggiati dai docenti a interpretare i concetti in modo personale e a sviluppare giudizi concettuali alternativi. Tuttavia, l'informatica è una disciplina scientifica e possiede molti concetti definiti e implementati con precisione e rigore scientifico. Ci si aspetta, dunque, che gli studenti raggiungano caratteristiche ben definite di comprensione di un concetto, come ad esempio cosa fa l'istruzione di assegnazione in Java, oppure cosa è un oggetto e come viene trattato durante l'esecuzione. A volte un programmatore alle prime armi non comprende pienamente un concetto, e conseguentemente ne dà una interpretazione errata, anche se involontariamente. La comprensione errata e incompleta dei concetti di programmazione si traduce in scrittura di codice errato che porta, inevitabilmente, lo studente alla creazione di programmi non funzionanti e scorretti. Sfortunatamente, idee sbagliate anche sui concetti di programmazione più fondamentali, che sono banali per gli esperti, sono comuni tra i principianti e difficili da superare.

Sorva [26] considera molti tipi di difficoltà cognitive intese come misconcezioni, riconoscendo che ci sono differenze qualitative tra i vari tipi di difficoltà che gli studenti possono incontrare. Per gli studenti di informatica che studiano i cicli, ad esempio, queste difficoltà potrebbero variare da errori di sintassi associati alle istruzioni di ripetizione, oppure a incomprensioni sull'esecuzione di un ciclo che possono riguardare la errata concezione del costrutto stesso. Oltre alle difficoltà di sintassi o di comprensione dell'esecuzione, lo studente potrebbe avere difficoltà nell'utilizzo stesso del costrutto per risolvere un problema. Le misconcezioni, dunque, secondo Sorva sono probabilmente meglio definite come errori nella comprensione concettuale. Tuttavia, le difficoltà che gli studenti incontrano nella programmazione non sono sempre chiaramente identificabili e le misconcezioni possono contribuire ad altri tipi di difficoltà o errori che gli studenti

incontrano durante la fase di apprendimento della programmazione.

I fattori che possono contribuire alla creazione delle misconcezioni

Le misconcezioni sono presenti in quasi tutti gli studenti. In letteratura sono presenti molti studi sulle misconcezioni. Qian e Lehmann [27] hanno fornito un'ampia revisione della letteratura sulle misconcezioni legate alla programmazione. Secondo gli autori, una serie di fattori che possono contribuire alle misconcezioni e ad altre difficoltà incontrate dagli studenti, possono essere i seguenti.

- **Complessità dell'esercizio ed eccessivo carico cognitivo**: La complessità dell'esercizio o del compito può contribuire alla creazione delle misconcezioni degli studenti nella programmazione aumentando il carico cognitivo e portando allo smarrimento. Dato che i principianti che stanno imparando a programmare non hanno familiarità con tutti i nuovi termini e la sintassi del linguaggio di programmazione, possono dimenticare alcune parti sintattiche di base come parentesi, parentesi graffe, punti, punti e virgola. Oppure possono, ad esempio, confondere gli operatori di base mentre scrivono semplici programmi. La complessità eccessiva dei programmi, inoltre, porta lo studente a fare errori più frequentemente soprattutto all'aumentare della complessità dei concetti teorici. Ne è una dimostrazione uno studio effettuato da Sanders e Thomas [28] su studenti di un corso di informatica di base. Sanders e Thomas hanno, infatti, evidenziato attraverso il loro studio che la maggior parte degli studenti è in grado di svolgere esercizi e scrivere programmi perfettamenti funzionanti nelle fasi iniziali del corso, ma quando gli esercizi e i programmi da svolgere diventano più impegnativi, gli studenti iniziano a fare molti errori.

- **Linguaggio naturale**: In particolare per gli studenti di lingua inglese potrebbero esserci dei motivi aggiuntivi che portano a misconcezioni dovute al linguaggio naturale. Infatti, dato che i comandi del linguaggio di programmazione sono basati sul linguaggio naturale della lingua inglese, la comprensione del linguaggio di programmazione da parte degli studenti può interferire con la loro comprensione dei significati dei comandi. Ad esempio alcuni studenti potrebbero pensare che

l'istruzione if sia sempre in attesa, aspettando che la condizione booleana sia vera, analogamente all'uso, nel linguaggio naturale inglese, della parola "if".

- **Conoscenza della matematica**: Motivo di misconcezione nella programmazione può essere anche la conoscenza della matematica. Dato che gli studenti fin dalla scuola media hanno avuto anni di esperienza in matematica, la nuova sintassi usata nella programmazione entra in conflitto con la matematica convenzionale, diventando così un vero e proprio ostacolo alla comprensione di concetti (anche elementari), che porta inevitabilmente alla creazione di misconcezioni. Infatti, gli studenti alle prime armi spesso sono portati a confondere il concetto di assegnamento di valori a della variabili con il concetto di espressioni algebriche che, senza una buona conoscenza e comprensione delle nozioni base della programmazione, potrebbero sembrare la stessa cosa. Un altro esempio di misconcezione dovuta alla conoscenza della matematica potrebbe essere, ad esempio, il problema della divisione fra interi in linguaggi come C e Java. In questi linguaggi di programmazione, infatti, il risultato della divisione di due interi, ad esempio 1/2 , è zero, mentre in matematica il risultato sarebbe 0.5.

- **Modelli mentali errati**: I modelli mentali discussi sopra possono creare, a loro volta, misconcezioni. Ad esempio, a differenza degli errori sintattici, come le parentesi mancanti e il punto e virgola, che sono ovvi e facili da correggere, le idee sbagliate degli studenti che derivano da un modello mentale incompleto o impreciso sono nascoste e resistenti al cambiamento e portano alla creazioni di misconcezioni.

- **Pattern e strategie inadeguati**: La conoscenza della programmazione acquisita dagli studenti inesperti di solito non è organizzata in modelli ben definiti e completi. Infatti, gli studenti che conoscono la sintassi e comprendono i concetti possono ancora incontrare difficoltà nel risolvere i problemi di programmazione. Perché spesso possiedono una conoscenza frammentaria invece di schemi ben organizzati. Gli studenti a cui mancano strategie di programmazioni non riescono a ragione a livello astratto e ciò porta alla creazione di misconcezioni.

- **Fattori esterni**: Le misconcezioni possono anche derivare da fattori esterni. Infatti, sebbene molte idee sbagliate e difficoltà derivino dalle conoscenze esistenti

degli studenti, i fattori esterni, come le caratteristiche del linguaggio e l'ambiente di programmazione, possono anche rappresentare un ostacolo al successo degli studenti nella comprensione dei concetti di programmazione. Ad esempio, in alcuni linguaggi di programmazione, come Java, l'operatore di addizione (+) può essere utilizzato per aggiungere numeri interi, concatenare stringhe o anche concatenare interi con stringhe. Inoltre anche l'ambiente di programmazione potrebbe diventare oggetto di creazione di misconcezioni. Ad esempio, i principianti possono affrontare delle difficoltà quando utilizzano strumenti di debug di ambienti di programmazione che non supportano correttamente le attività di debug. Inoltre, i messaggi di errore criptici e poco informativi rendono difficile la correzione degli errori per i principianti.

- **Istruzione e conoscenza dei docenti**: Un ultimo fattore che può contribuire alla creazione di misconcezioni è il docente. A volte può essere l'insegnante stesso a innescare la creazione, inconsapevolmente, di modelli mentali errati negli studenti. Esempi di strategie di insegnamento problematiche includono l'uso di analogie, modelli e metafore inappropriati. Secondo molti studiosi, ad esempio, descrivere una variabile come una scatola, che è qualcosa che fanno molti insegnanti di corsi introduttivi alla programmazione, è un esempio di applicazione errata di analogie nella didattica della programmazione. Gli studenti, infatti, potrebbero credere che una variabile possa contenere un certo numero di cose proprio come una scatola. Sebbene l'uso di analogie possa aiutare a spiegare sistemi complessi, l'uso inappropriato di analogie può portare a misconcezioni strutturate che impediscono ai principianti di stabilire modelli mentali accurati.

2.3.5 Il carico cognitivo

Le teorie delle scienze cognitive hanno fortemente influenzato la letteratura della didattica, anche della didattica dell'informatica. Il concetto di *carico cognitivo* ha assunto negli anni sempre più rilevanza. La teoria del carico cognitivo si basa sul concetto che la memoria di lavoro può essere sottoposta a diversi "sforzi cognitivi" durante l'esecuzione di un'attività mentale. Questa teoria si basa sul funzionamento del cervello umano,

prendendo in considerazione sia la memoria di lavoro che la memoria a lungo termine, come accennato nel capitolo 1. L'apprendimento avviene quando nuove informazioni sono associate a conoscenze già archiviate ed organizzate nella memoria a lungo termine, producendo, in questo modo, una nuova conoscenza più elaborata ed estesa. Secondo Plass [29] il problema centrale della teoria del carico cognitivo è che l'apprendimento risulta essere compromesso quando la quantità di lavoro cognitivo richiesto eccede la capacità strutturale e fisiologica della memoria di lavoro. La teoria del carico cognitivo, come originariamente proposto da John Sweller [30], descrive tre differenti tipi di carico:

- Intrinseco: carico dovuto allo sforzo cognitivo e alla difficoltà di un compito specifico.
- Estraneo: carico derivante da fattori esterni, non indispensabile per l'esecuzione del compito e non necessario per imparare.
- Pertinente: effettivo carico cognitivo determinato dallo sforzo di apprendimento, ma non indispensabile per la soluzione del compito.

Dunque, l'apprendimento dipende fortemente dal grado di difficoltà cognitiva e dalla quantità di nuove informazioni da elaborare. Lo scopo del docente deve essere quello di progettare del materiale didattico che sia tecnicamente corretto ed esaustivo, ma, allo stesso tempo, deve essere predisposto affinché i concetti non siano in grado di sovraccaricare la memoria di lavoro.

Carico cognitivo nella programmazione

La didattica dell'informatica si è concentrata spesso sul concetto di carico cognitivo, in quanto, le attività di programmazione implicano tipicamente un elevata richiesta della memoria di lavoro e conseguentemente un elevato carico cognitivo. Come descritto nel capitolo precedente, l'uomo è capace di svolgere attività mentali complesse usando il raggruppamento, infatti, il numero di elementi (o anche definiti *chunck*) che possono essere immagazzinati simultaneamente nella memoria di lavoro sono generalmente pari a 4 ± 1. La programmazione richiede un grande carico di lavoro sulla memoria di lavoro. Molti studiosi della didattica dell'informatica hanno cercato di capire quanto il carico

cognitivo sia determinante nella fase dell'apprendimento della programmazione. In una serie di studi si è cercato di testare gli studenti utilizzando esempi pratici per verificare la conoscenza acquisita e le capacità di apprendimento. I risultati hanno mostrato come gli studenti che hanno studiato gli esempi pratici già forniti, hanno ottenuto risultati migliori rispetto a quelli che hanno semplicemente risolto problemi senza l'utilizzo di esempi.
Una delle strategie per diminuire il carico cognitivo durante la fase dell'apprendimento della programmazione è quella di utilizzare esempi parzialmente elaborati che vengono completati dallo studente in più fasi. Gray et al. [31] hanno presentato una serie dettagliata di suggerimenti per l'implementazione di esempi incompleti per un corso introduttivo alla programmazione in C++, componendo un compito di programmazione in componenti il cui carico cognitivo può essere adeguatamente gestito dagli studenti. Gli autori sostengono che questi tipi di esempi aggiungono un'esposizione graduale e ripetuta dei concetti nell'apprendimento della programmazione attraverso l'elaborazione di una varietà di problemi. L'esposizione ripetuta a una varietà di problemi e il fatto che gli studenti giustifichino i loro passaggi risolutivi aiutano a sviluppare abilità di apprendimento che sono essenziali per costruire uno schema efficace. I risultati dell'apprendimento sono migliorati grazie all'utilizzo di strumenti che aiutano la visualizzazione del programma e la costruzione di modelli mentali accurati. Tuttavia, come dimostrato dagli studi analizzati, la ricerca sul carico cognitivo suggerisce un imperativo di progettazione didattica della programmazione: la necessità di mantenere gli strumenti di programmazione per principianti il più semplici possibile in modo da ridurre il carico cognitivo al fine di facilitare il processo di apprendimento.

2.4 Strategie per la didattica della programmazione

Se per uno studente che si approccia al mondo della programmazione imparare a programmare può essere molto difficile dobbiamo inevitabilmente porci una serie di domande fondamentali per l'apprendimento di questa materia: per uno studente novizio quanto è importante la conoscenza teorica? Quanto conta la pratica? Quali strategie è possibile insegnare ad uno studente che si approccia per la prima volta alla programmazione?

Secondo Davies [32] imparare a programmare implica l'acquisizione sia di conoscenze teoriche, come ad esempio essere in grado di affermare come un ciclo for funziona, sia di conoscenze pratiche utili all'applicazione dei concetti teorici, come ad esempio utilizzare il ciclo for in un contesto appropriato. Tuttavia, Davies sostiene che molti corsi di studi sulla programmazione di base si concentrano particolarmente sulla parte descrittiva e teorica dei concetti tralasciando, spesso, la parte pratica, fondamentale per la comprensione e l'applicazione dei concetti. Nei paragrafi precedenti si è parlato delle principali difficoltà cognitive che uno studente alle prime armi può imbattersi durante l'apprendimento della programmazione. Il compito del docente è fare sì che lo studente sia guidato in tutto il processo di apprendimento affinché possa comprendere pienamente i concetti. Individuare strategie per rendere la didattica più efficiente è dunque un passaggio fondamentale ed inevitabile per qualunque docente di informatica.

2.4.1 Il ruolo del docente nell'insegnamento della programmazione

Gli insegnanti dovrebbero concentrare i propri sforzi sulle prospettive fondamentali che trasformano la visione della disciplina dello studente, piuttosto che introdurre concetti teorici all'interno di un vasto agglomerato di argomenti dove possono passare quasi inosservati. La ricorsione e l'istanziazione degli oggetti, ad esempio, sono concetti abbastanza difficili (come gli studi osservati in precedenza hanno evidenziato) da comprendere in particolare per un principiante che è in grado di pensare ai programmi in termini poco astratti e di tracciarne l'esecuzione passo dopo passo. Senza la "lente" delle dinamiche della programmazione, che solo il docente è in grado di insegnare agli studenti, padroneggiare questi importanti concetti diventa molto difficile. Inoltre, un insegnante di informatica di base non deve cadere nella trappola di presumere che gli studenti pensino e agiscano proprio come fa lui. Secondo Sorva [25] infatti, gli insegnanti devono aiutare gli studenti a scoprire la natura del "gioco" sottostante alla programmazione. Uno studente che supera un corso di programmazione di base senza aver sviluppato una prospettiva dinamica sull'esecuzione del programma non ha davvero imparato molto sulla programmazione, indipendentemente da quante definizioni concettuali possa aver memorizzato o da quanti modelli di codice possa essere in grado di applicare. Al contrario, coloro i

quali hanno imparato, anche grazie all'esperienza del docente, quello che Sorva definisce il "gioco sottostante" delle programmazione, saranno incentivati ad imparare sempre di più e con più facilità.

2.4.2 Favorire la realizzazione di modelli mentali corretti e della macchina concettuale

Secondo Sorva [25] l'insegnamento esplicito di una macchina concettuale potrebbe diminuire il livello di libertà che gli studenti si concedono mentre formano modelli mentali, portando lo studente alla creazione di modelli mentali giusti. Infatti, è importante che i programmatori inesperti sviluppino modelli mentali giusti per imparare a programmare e per comprendere i concetti chiave teorici. Ogni corso introduttivo alla programmazione prevede una macchina concettuale; la macchina è implicita nel linguaggio di programmazione utilizzato. Il docente deve rendere l'apprendimento di una macchina concettuale un obiettivo esplicito piuttosto che implicito, ovvero attraverso una spiegazione elaborata allo scopo di illustrare la struttura di un sistema e il funzionamento interno nella fase di insegnamento della programmazione. Come si è discusso dall'analisi degli studi presi in considerazione fino ad ora, secondo i ricercatori della teoria dei modelli mentali, è utile cercare di favorire la realizzazione di modelli mentali corretti. Un modello mentale corretto può essere insegnato anche solo attraverso una semplice metafora o analogia, o una spiegazione più complessa del sistema. Infatti, ciò è in linea con le scoperte dei ricercatori sui modelli mentali secondo cui cambiare un modello iniziale tende ad essere più difficile che crearne uno nella fase iniziale dell'apprendimento. Lo scopo del docente deve essere, dunque, quello di evitare che lo studente crei un modello mentale scorretto. Lo studente che si è creato una macchina concettuale errata, infatti, potrebbe essere convinto di saper spiegare come avviene il comportamento di alcuni programmi, anche se questi generalmente sono errati. Un modello mentale errato altera ulteriormente la convinzione dello studente della propria comprensione relativa ad un argomento. Sebbene gli studenti, ovviamente, non si presentino come "contenitori vuoti" da riempire di concetti, individuare modelli mentali errati e correggerli è la prima cosa da fare per un didattica efficace ed efficiente. Un modello mentale corretto può rendere espliciti i modi in cui il comportamento della macchina differisce dal pensiero umano e può sotto-

lineare come un linguaggio di programmazione differisce dal linguaggio naturale e dalla matematica. Inoltre, un modello di una macchina concettuale può servire come base per insegnare il tracciamento (tracing) del programma, di cui siparlerà nel paragrafo successivo. Il modello mentale, dunque, può suggerire agli studenti una prospettiva adeguata da adottare mentre si "tracciano" i programmi.

2.4.3 Tracing e debug

Uno dei migliori metodi per insegnare corretti modelli mentali e valide macchine concettuali è applicare il tracciamento del programma, durante la fase di apprendimento dei concetti della programmazione. In letteratura sono presenti molti studi che evidenziano come la fase di tracing possa essere estremamente utile all'apprendimento della programmazione, al fine di evitare che gli studenti creino modelli mentali errati che si ripercuotono per tutta la durata del corso di programmazione. Di conseguenza, i modelli visivi sono stati studiati come un modo per aiutare gli studenti novizi. Vainio e Sajaniemi [33] hanno effettuato uno studio sull'importanza del tracciamento durante l'apprendimento della programmazione. Lo studio si basava su una serie di interviste con studenti che hanno partecipato a un corso estivo intensivo sulla introduzione alla programmazione. Il linguaggio di programmazione utilizzato era Java e il corso copriva la programmazione elementare dai costrutti del linguaggio di base, come l'iterazione, agli oggetti, le interfacce, le classi astratte e l'ereditarietà e semplici algoritmi per l'ordinamento e la ricerca. Ogni studente è stato intervistato una volta alla settimana durante il corso, realizzando un totale di 32 interviste. Le interviste contenevano domande aperte, definizioni di concetti, attività di tracciamento del programma e compiti di programmazione risolti con carta e penna. Le interviste sono state registrate e trascritte per un'analisi qualitativa in cui il comportamento di tracciamento degli studenti è stato accuratamente confrontato con l'esecuzione effettiva del programma. Lo studio ha valutato principalmente il semplice tracciamento di un singolo valore. Questa strategia di valutazione è caratterizzata principalmente due categorie di variabili nei programmi: quelle a cui è assegnato un valore che può essere visto direttamente nel codice del programma (ad esempio, "a = 15") e quelle che sono i risultati di alcuni calcoli e i cui valori non possono essere visti nel programma stesso. Gli autori hanno osservato che lo studente, generalmente, ha grandi

difficoltà di tracciamento. Le difficoltà individuate, secondo gli autori, hanno le loro radici sia nella mancanza di esperienza che nella mancanza di comprensione concettuale della programmazione, che richiedono diversi tipi di sforzi didattici.
Matthew e Jump [34] hanno rilevato scarsi rendimenti degli studenti durante due corsi di introduzione alla programmazione. Questi problemi si sono verificati nonostante i ricercatori abbiano apportato una serie di modifiche pedagogiche, inclusa l'aggiunta di esercizi di apprendimento attivo ed esempi già sviluppati. Gli autori hanno inoltre notato che gli studenti sembravano avere una particolare difficoltà con problemi come l'assegnamento delle variabili. Inoltre, la maggior parte degli studenti ha avuto difficoltà a scrivere metodi ricorsivi. Le lezioni includevano già attività di tracciamento del codice in classe, laboratori, piccoli esercizi di programmazione, progetti di programmazione più ampi e altre opportunità per gli studenti di sviluppare validi modelli mentali. Tuttavia, gli autori hanno osservato che dal momento in cui è stata posta particolare importanza alla fase di tracciamento del codice, gli studenti hanno fornito agli insegnanti un punto di partenza molto più chiaro con cui discutere sui modelli corretti. Inoltre, da un test effettuato, è risultato che gli studenti sono stati estremamente positivi sull'utilizzo del tracciamento per spiegare i concetti. La maggioranza degli studenti di ogni classe ha risposto che il tracciamento del codice mostrato ed illustrato dall'insegnante durante la lezione è stato "molto utile" per l'apprendimento del materiale. Inoltre, molti studenti hanno anche riferito che il tracciamento spiegato dai docenti li ha aiutati a imparare in modo più efficace e incisivo.

2.4.4 Il miglior linguaggio di programmazione per imparare la programmazione di base

L'obiettivo del corso di base di programmazione è quello di introdurre gli studenti alla programmazione e al problem solving utilizzando un qualsiasi linguaggio di programmazione. L'obiettivo dell'insegnamento della programmazione è molto semplice: gli studenti devono imparare a programmare utilizzando paradigmi utili alla comprensione della programmazione e con cui, con grande probabilità, avranno spesso a che fare nel loro percorso di studi o in ambito lavorativo (come ad esempio il paradigma della programmazione strutturata o quello della programmazione orientata agli oggetti). Il

linguaggio deve dunque essere unico e semplice, perché lo scopo è imparare le basi della programmazione e apprendere i concetti di base. Imparare i concetti teorici di base è fondamentale, in quanto tali concetti devono essere facilmente trasferibili ad altri linguaggi di programmazione. Lo specifico linguaggio di programmazione adottato deve essere solo, dunque, il mezzo attraverso il quale gli studenti mettono in pratica le nozioni teoriche acquisite.

La letteratura esaminata fino ad ora include studi basati sulla progettazione di corsi che sono specifici per un certo tipo di linguaggio di programmazione e hanno dimostrato di avere un effetto benefico sui risultati di apprendimento. Tra questi vi è sicuramente la ricerca e l'uso di linguaggi sintatticamente semplici e di ambienti e strumenti di programmazione ricchi di funzionalità. Inoltre, come già visto in precedenza, è importante individuare linguaggi che incoraggiano la creazione di una macchina concettuale corretta e pongono grande attenzione sia alla fase di tracing che alla scrittura stessa del codice. Secondo molti studiosi, oggi Python può essere considerato come un linguaggio estremamente adatto alla didattica nei corsi introduttivi alla programmazione. Python è stato progettato nel 1991 da Guido van Rossum proprio per risultare un linguaggio alla portata di tutti, intuitivo ed immediato. Lo dimostra, tra l'altro, la semplicità della sua sintassi. Secondo Goldwasser e Letscher [35], infatti, Python si adatta perfettamente anche alla didattica della programmazione ad oggetti. Stefik e Siebert [36] hanno studiato l'impatto della sintassi dei linguaggi di programmazione tra programmatori esperti e programmatori novizi tramite questionari. In questo studio, i costrutti del linguaggio di programmazione (dichiarazioni di variabili, assegnazione, cicli, condizioni e così via) di un insieme di linguaggi di programmazione diversi sono stati presentati a programmatori principianti ed esperti, che dovevano valutare quanto fossero intuitive le costruzioni della sintassi. Per i programmatori esperti, costruzioni simili a quelle che già conoscono sono, come previsto, più intuitive. Gli esercizi da risolvere erano semplici compiti in sei linguaggi di programmazione: Quorum, Perl, Java, Ruby, Python e Randomo (un linguaggio falso creato generando casualmente alcune parole chiave). Hanno scoperto che, per quanto riguarda gli studenti novizi, per i linguaggi con una sintassi simile al C (Java e Perl), i risultati di accuratezza non erano statisticamente e significativamente diversi da quelli di Randomo (che svolge il ruolo di un linguaggio di programmazione placebo),

mentre l'accuratezza per Python era molto più alta. In questo modo, gli autori hanno mostrato che molti aspetti della sintassi tradizionale in stile C, sebbene abbia influenzato una generazione di programmatori, presenta problemi in termini di comprensibilità per i principianti.

2.4.5 La conoscenza pregressa della programmazione degli studenti

Per comprendere la conoscenza pregressa della programmazione da parte degli studenti, è stata effettuata un'interessante indagine quantitativa dai docenti del corso. La scelta del primo paradigma di programmazione e del linguaggio attraverso il quale dovrebbe essere introdotta la programmazione agli studenti novizi è un tema di ricerca centrale nella didattica della programmazione. Alexandron et al. [37] hanno studiato l'influenza dell'esperienza passata e della conoscenza acquisita nell'ambito della programmazione. I risultati hanno mostrato che per quanto riguarda il livello di astrazione, il passaggio da parte di studenti che imparano un nuovo linguaggio di programmazione, da uno di basso livello ad uno di alto livello, può portare i programmatori novizi a sentire di perdere potere e controllo del programma. Inoltre, lavorare ad un livello di astrazione elevato significa concentrarsi di più su ciò che fa il programma e meno su come lo fa. I risultati mostrano anche che l'influenza dovuta all'esperienza nei linguaggi procedurali e sequenziali porta gli studenti ad interpretare il nuovo modello mentale attraverso il prisma dei modelli precedenti, con cui hanno già familiarità. Dunque gli studenti cercano attivamente di forzare il nuovo modello a comportarsi come il modello con cui hanno familiarità, in modo che possano utilizzare soluzioni di programmazione precedentemente acquisite. Secondo lo studio, infatti, la perdita della sensazione di controllo sul programma da parte di studenti che imparano un nuovo linguaggio ad alto livello può indurli a sviluppare un atteggiamento negativo che può influenzare le loro prestazioni e che si può ripercuotere per l'intero processo di apprendimento del nuovo linguaggio.
Wilcox e Lionelle [38]] hanno cercato di quantificare i benefici della conoscenza pregressa nella programmazione fornendo, inoltre, attenzione alle differenze di genere. La ricerca si basa su un corso di programmazione introduttiva in cui gli studenti sono stati suddivisi in due gruppi: uno con studenti con precedenti esperienze di programmazione e uno

con studenti senza alcuna esperienza pregressa. I risultati suggeriscono che i vantaggi conferiti dall'esperienza pregressa della programmazione sono statisticamente significativi. Anche per quanto riguarda i risultati finali, lo studio ha evidenziato che studenti che hanno avuto un'esperienza passata con la programmazione hanno ottenuto maggiori risultati positivi rispetto agli studenti senza esperienza, dimostrando in questo modo che i vantaggi dell'esperienza precedente sono sostanziali. I risultati infatti mostrano che gli studenti con esperienza pregressa superano gli studenti senza alcuna esperienza di oltre il 6% agli esami e del 10% ai quiz di programmazione. Lo studio, inoltre, suggerisce un interessante spunto per quanto riguarda la disparità di genere. Infatti, gli autori sottolineano il netto contrasto tra studenti e studentesse, dimostrando il totale sbilanciamento tra i due generi in termini di conoscenza pregressa della programmazione. I dati hanno mostrato, infatti, una maggiore percentuale di studenti maschi con conoscenze pregresse di programmazione rispetto agli studenti di genere femminile.
Si potrebbe dunque pensare che possa essere fondamentale in alcuni casi individuare precocemente quali studenti abbiano delle conoscenze pregresse e quali no, al fine di dividere il corso in diverse parti per garantire agli studenti con poca esperienza di essere al passo di quelli con esperienza pregressa. Kirkpatrick e Mayfield [39] hanno studiato il comportamento degli studenti con differenti conoscenze di background sulla programmazione, in quanto è un pensiero comune che una classe eterogenea, in termini di esperienza sulla programmazione, possa sollevare la preoccupazione negli studenti novizi rendendo il contesto didattico sempre più "intimidatorio". Per determinare se queste preoccupazioni fossero reali, i ricercatori hanno suddiviso il corso per studenti senza esperienza in più corsi per facilitare l'apprendimento. I risultati mostrano che la sequenza complessiva di più corsi riesce a consentire agli studenti con poca o nessuna esperienza di programmazione di mettersi al passo con i loro coetanei con esperienza precedente nel momento in cui raggiungono corsi di informatica avanzati.

Capitolo 3

Il corso di informatica per studenti del primo anno di matematica dell'Università di Bologna

Gli studi analizzati nei precedenti capitoli hanno evidenziato le principali difficoltà che gli studenti novizi posso avere durante il processo di apprendimento della programmazione. Nel capitolo 2 si è discusso, in particolare, delle difficoltà cognitive tipiche di chi impara per la prima volta a programmare, ponendo grande attenzione alle possibili cause delle difficoltà cognitive e ad eventuali accorgimenti che gli insegnanti possono prendere per rendere la didattica più efficace.
I dati a disposizione per questa tesi riguardano al corso di Informatica per studenti del primo anno del corso di laurea in Matematica dell'Università di Bologna. Il target del seguente studio sono, dunque, studenti del primo anno di Matematica che imparano a programmare con il linguaggio di programmazione Python. Nel seguente capitolo si tratterà, nello specifico, del corso oggetto di analisi di questa tesi, e degli argomenti trattati durante la parte di laboratorio di programmazione. Si procederà, successivamente, a mostrare le principali caratteristiche dell'ambiente di sviluppo utilizzato durante il laboratorio e i log generati da esso. In conclusione si definirà l'obiettivo dell'analisi ponendo grande attenzione alla struttura dei dati.

3.1 Il corso di informatica e il laboratorio di programmazione

Il corso di laurea triennale in Matematica dell'Università di Bologna è un corso di studi universitario di primo livello della durata di 3 anni. Il secondo semestre del primo anno include, come attività obbligatoria, il corso di Informatica. Il corso di informatica preso in analisi è di 8 crediti formativi ed è stato svolto durante il secondo semestre del 2020 dal Professor Simone Martini. Il corso è stato svolto con la collaborazione di tre tutor (relatore e correlatori di questa tesi). A causa della pandemia dovuta al virus COVID-19 il corso si è tenuto interamente a distanza. Tra le conoscenze e le abilità che il corso permette di conseguire agli studenti di matematica vi è l'acquisizione del necessario background informatico ed un'appropriata conoscenza dei principali costrutti di un linguaggio di programmazione. Lo scopo del corso è quello di insegnare agli studenti a saper utilizzare le conoscenze acquisite per progettare autonomamente algoritmi e strutture dati. Il corso, inoltre, prevedeva una parte di laboratorio nel quale sono stati trattati argomenti relativi alla programmazione. Come linguaggio di programmazione è stato scelto Python. Come anticipato nel capitolo 2 (vedi paragrafo 2.4.4) Python può essere considerato come un linguaggio estremamente adatto alla didattica di corsi introduttivi alla programmazione.

Il corso è stato interamente svolto sulla piattaforma piattaforma Microsoft Teams. A tutte le lezioni e ai laboratori hanno partecipato il professore principale e tutti gli assistenti. Durante le lezioni uno degli assistenti partecipava in chat per sottolineare alcuni dei concetti principali presentati, o indicare materiale trattato in precedenza. Gli studenti avevano la possibilità di porre domande anche in chat. Tutte le domande hanno avuto risposta dal professore di quella lezione, ad eccezione di quelle tecniche o banali, alle quali è stata data risposta direttamente in chat dal primo tutor disponibile. Dopo aver assegnato le esercitazioni durante i laboratori, i tutor sono rimasti online per un po' di tempo per rispondere ad altre domande o discutere con gli studenti. Dopo le lezioni, il docente e i tutor hanno avuto lunghi debriefing.

I dati acquisiti ed analizzati per questo elaborato comprendono i log generati dall'ambiente di sviluppo Thonny utilizzato da tutti gli studenti partecipanti al corso. I dati

sono stati anonimizzati. Gli studenti del corso che hanno acconsentito alla raccolta di log sono stati circa 100. Ad ogni laboratorio venivano proposti alcuni esercizi da svolgere e gli studenti li risolvevano su Thonny. Per ogni esercizio assegnato i docenti fornivano supporto in chat privata (essendo le lezioni in modalità a distanza) a chi lo chiedesse. Dopo 20-30 minuti l'esercizio veniva corretto per tutti gli studenti e in seguito si passava all'esercizio successivo.

3.1.1 Lezioni e argomenti trattati durante il laboratorio di programmazione

Durante il corso, i laboratori sono stati 14. Ogni lezione ha avuto una durata di 2 ore. Gli argomenti trattati durante le lezioni di laboratorio sono stati rispettivamente:

- Introduzione all'uso di Python
- Variabili, assegnamento, funzioni
- Booleani, selezione, input
- Stringhe, tuple, e iterazioni su sequenze
- Iterazione determinata (uso di range)
- Debugging
- Iterazione indeterminata
- Liste
- Ricorsione
- Dizionari
- List Comprehensions
- Classi e oggetti
- Ereditarietà

- Alberi binari

Ad ogni lezione di laboratorio venivano assegnati alcuni esercizi sugli argomenti trattati durante le lezioni teoriche svolte nei giorni precedenti. Alla fine della lezione venivano assegnati degli esercizi da svolgere a casa.

3.2 I risultati dell'indagine

I risultati dell'indagine effettuata dai docenti del corso mostrano un numero consistente di studenti con nessuna esperienza di programmazione. In particolare:

- 77 studenti hanno dichiarato di non avere precedenti esperienze;
- 26 studenti hanno dichiarato di aver studiato programmazione al liceo, di cui 1 afferma di averla studiata solo superficialmente;
- 4 studenti hanno dichiarato di aver seguito un corso di programmazione in contesti extra scolastici;
- 2 studenti hanno dichiarato di aver studiato la programmazione da autodidatta;
- 1 studente ha dichiarato di aver studiato programmazione al liceo e da autodidatta, studiando diversi linguaggi;

Sono state, inoltre, acquisite informazioni sui percorsi di studi superiori degli studenti. Non essendoci una omogeneità nei percorsi di studi di tutto il mondo, è molto difficile trovare studi che dimostrino l'effetto che può avere un percorso di studi superiore rispetto ad un altro nel contesto della programmazione. Essendo il corso italiano, i dati raccolti si riferiscono a percorsi del sistema scolastico italiano. Dall'indagine è emerso che dei 25 studenti che hanno studiato la programmazione al liceo:

- 16 studenti hanno dichiarato di provenire dal liceo scientifico con indirizzo "scienze applicate", nel quale si studia informatica per 2 ore alla settimana;
- 3 studenti hanno dichiarato di provenire dal liceo scientifico, ma non hanno dichiarato quale indirizzo. Tuttavia, è molto probabile che anche questi studenti abbiano

frequentato l'indirizzo scienze applicate in quanto nel Liceo scientifico tradizionale non l'informatica non viene insegnata;

- 2 studenti hanno dichiarato di provenire dal "liceo tecnologico scientifico", un vecchio indirizzo sperimentale, molto simile all'indirizzo "scienze applicate";
- 3 studenti hanno dichiarato di provenire dall'indirizzo "sistemi informativi aziendali" degli istituti tecnici settore economico;
- 2 hanno dichiarato di provenire dall'indirizzo "informatica" degli istituti tecnici settore tecnologico;

Per quanto riguarda il linguaggio di programmazione, sono stati raccolti anche dati sui linguaggi conosciuti dagli studenti con precedenti esperienze di programmazione. Lo stesso studente poteva dichiarare di conoscere più linguaggi. I linguaggi conosciuti dagli studenti sono mostrati nella tabella 3.1.

Linguaggio	Frequenza	Linguaggio	Frequenza
C++	24	C	13
Scratch	9	Basic/VB	8
Java	7	Pascal	7
Python	7	SQL	6
PHP	5	C#	4
HTML	4	App Inventor	3
Javascript	3	Assembly	2
CSS	1	Snap!	1

Tabella 3.1: Linguaggi utilizzati dagli studenti prima del corso

Per 93 studenti il 2020 è stata la prima volta che hanno frequentato e seguito il corso, mentre 17 hanno seguito il corso negli anni precedenti. Agli studenti è stato anche chiesto di valutare il numero di lezioni di laboratorio che hanno seguito, la risposta prevedeva di scegliere un numero fra 0 e 5, dove 0 corrispondeva a non aver frequentato per nulla, mentre 5 ad aver frequentato tutti i laboratori. La frequenza per il corso non era obbligatoria. Sono stati effettuati questionari per l'intero corso, ma i dati che

interessano questo lavoro sono quelli riguardanti la frequenza per i laboratori,riassunti dalla tabella 3.2.

Valutazione	0	1	2	3	4	5
Frequenza	6	0	2	4	13	85

Tabella 3.2: Valutazione sul numero dei laboratori seguiti dagli studenti

3.3 Thonny: un ambiente di sviluppo per Python open source

Per quanto riguarda la parte di laboratorio, come ambiente di sviluppo per la didattica della programmazione è stato scelto Thonny. Thonny è un ambiente di sviluppo per Python progettato specificamente per i novizi. Oltre ad essere un IDE semplice da utilizzare e da configurare è un progetto open source e gratuito. Data la sua interfaccia minimale, Thonny rende la fase di avvicinamento ad un ambiente di sviluppo più facile essendo, per l'appunto, pensato per offrire unicamente le opzioni essenziali, in modo tale da non distrarre eccessivamente lo studente con funzionalità che potrebbero risultare superflue e non adatte alla fase iniziale di apprendimento della programmazione.
Thonny è un progetto sviluppato da Aivar Annamaa, il quale in [40] e [41] ne ha illustrato le principali caratteristiche, definendolo un ambiente di sviluppo Python per l'apprendimento e l'insegnamento della programmazione capace di rendere naturale la visualizzazione del programma e parte del flusso di lavoro per gli studenti novizi. Infatti, tra le caratteristiche principali di Thonny vi è la parte di visualizzazione del codice. I programmi scritti dallo studente possono essere facilmente eseguiti in modalità stepping, in cui l'esecuzione viene sospesa prima di eseguire ogni istruzione in modo da valutare step dopo step il comportamento del codice. Nel capitolo 2 è stato ampiamente discusso sull'utilità della fase di tracciamento del codice, Thonny è infatti un ambiente di sviluppo che si adatta perfettamente al tracing. Inoltre, è anche adatto alla fase di debug del codice in quanto è presente un pratico debugger che evidenzia i problemi del codice. Aivar Annamaa sostiene che l'interfaccia utente di Thonny sia adatta principalmente ai

principianti, in [41] ha dimostrato che la maggior parte degli studenti che hanno deciso di installare Thonny sui propri computer sono stati in grado di utilizzare le sue funzionalità senza ulteriore assistenza da parte dei docenti. Inoltre, l'autore ha evidenziato che durante un semestre di utilizzo di Thonny nel contesto del primo corso di programmazione in università, ha ricevuto feedback incoraggianti dagli studenti, i quali hanno ritenuto che le animazioni del tracciamento del programma presentate con Thonny li aiutassero ad apprendere nuovi costrutti di programmazione. La maggior parte degli studenti che ha utilizzato Thonny, infatti, durante il semestre del corso di programmazione ha affermato di essere stato aiutato, oltre che dall'interfaccia facile ed intuitiva e dalle funzionalità di tracciamento del codice, anche dalla parte del debugger.

3.3.1 Log di Thonny

Una caratteristica molto interessante di Thonny è quella della raccolta automatica dei log. I laboratori del corso sono stati 14. A partire dal quinto laboratorio i docenti hanno chiesto agli studenti di fornire loro i log generati da Thonny. Ad ogni lezione venivano assegnati degli esercizi che dovevano essere risolti durante le ore di laboratorio. Thonny ha dunque registrato tutto ciò che gli studenti effettuavano nell'ambiente di sviluppo durante l'esecuzione degli esercizi. I log vengono registrati su file. Ogni file nella directory dei log dell'utente rappresenta una sessione. La sessione corrisponde all'avvio della GUI di Thonny fino alla sua chiusura. I log sono strutturati in un array JSON. Ogni elemento nell'array JSON rappresenta un evento. Dunque ciascun file conterrà una lista di eventi che corrispondono ad un determinato evento nell'ambiente di sviluppo ad un certo tempo t. Il nome dell'evento è nel campo "sequence".

3.4 Dati raccolti

Tra tutti gli studenti che hanno partecipato al corso, 100 hanno acconsentito alla raccolta dei log di Thonny. I docenti hanno raccolto i dati a partire dal laboratorio cinque. Dunque i log a disposizione appartengono a 100 studenti per circa nove laboratori. Inoltre è anche possibile accedere, sempre in forma anonima, ai voti che ciascuno studente che ha acconsentito alla raccolta dei log ha ottenuto all'esame finale. Dato che ad ogni

laboratorio venivano assegnati più esercizi, nello stesso file di log possono esserci sequenze di eventi relativi a diversi esercizi svolti nella stessa giornata. Inoltre per fare sì che i log vengano generati correttamente, agli studenti è stato chiesto alla fine di ogni lezione di laboratorio di chiudere l'ambiente di sviluppo e poi riaprirlo per scaricare i log. Per questo motivo, potrebbe capitare che nei dati non siano presenti log con la data e l'ora corretta, dovuti al fatto che gli studenti non abbiano seguito passo dopo passo le istruzioni date dai docenti. Inoltre, agli studenti venivano assegnati degli esercizi per casa. Gli esercizi per casa, con molta probabilità, venivano eseguiti su Thonny. Questo potrebbe essere un altro motivo per cui i dati presenti nei log non appartengano tutti solo agli esercizi svolti durante le ore di laboratorio. Può essere molto probabile, infatti, che nei log vi siano sequenze di eventi non appartenenti agli esercizi svolti durante il laboratorio e dunque non utili ai fini di quest'analisi. Questi problemi verranno risolti nelle fasi successive.Verranno specificati nel capitolo 4 i passaggi effettuati per pulire i dati acquisiti.

3.5 Lezioni di laboratorio per i quali sono stati raccolti i log

I log sono stati raccolti a partire dal laboratorio 4 fino all'ultima lezione di laboratorio (lab 14). Di seguito sono riportate tutte le lezioni di laboratorio di cui si possiedono i log. Queste informazioni sono molto utili per leggere i capitoli successivi e comprendere in modo più approfondito l'analisi. Le lezioni di laboratorio possono dunque essere riassunte come descritto nella tabella 3.3.

Tabella 3.3: Laboratori con data e argomento trattati su cui sono stati raccolti i log.

Data	Numero Lab	Argomento
19/03/2020	4	Stringhe, tuple e iterazioni su sequenze
23/03/2020	5	Iterazione determinata (uso di range) e turtle
26/03/2020	6	Debugging
30/03/2020	7	Iterazione indeterminata
06/04/2020	8	Liste

Continuation of Table 3.3		
Data	Numero Lab	Argomento
16/04/2020	9	Ricorsione
27/04/2020	10	Dizionari
30/04/2020	11	List comprehension
07/05/2020	12	Classi e oggetti
11/05/2020	13	Ereditarietà
14/05/2020	14	Alberi binari

Capitolo 4

Analisi dei log generati durante il corso

In questo capitolo si discute delle scelte implementative per l'analisi dei log. Dopo una breve introduzione sull'obiettivo dell'analisi per questa tesi, si illustrano le fasi compiute per la realizzazione di questa analisi e si descrivono gli strumenti e librerie utilizzate. Verranno poi trattate nello specifico le scelte implementative effettuate per l'analisi, descrivendo nel dettaglio ogni decisione presa per la sua realizzazione.

4.1 Introduzione

Come ben discusso in [1] è importante evidenziare che empiricamente, metodologicamente e teoricamente, i processi di progettazione di uno studio sulla didattica della programmazione non sono intrinsecamente diversi dai numerosi altri campi da cui la ricerca sulla didattica dell'informatica prende in prestito le sue teorie, metodologie ed epistemologie empiriche. Ciò che è diverso è la conoscenza e le strategie che i ricercatori devono utilizzare per progettare uno studio di successo. Nella progettazione di studi sull'apprendimento e l'insegnamento dell'informatica, i prototipi di studi che devono essere realizzati sono tutti appartenenti ad uno specifico dominio.
In questo caso individuare ed identificare una domanda di ricerca è stato il primo passo. L'analisi dei log può fornire informazioni molto interessanti sul comportamento degli

studenti durante la fase di laboratorio di programmazione. Gli eventi temporali possono fornire, inoltre, informazioni utili sulla base del tempo e della progressione del processo di apprendimento.

4.1.1 Analisi dei dati nella didattica della programmazione

In un contesto di ricerca, la raccolta e l'analisi di dati sono attività molto importanti. Attraverso l'analisi è possibile estrarre della conoscenza fondamentale per dimostrare ed individuare nuove metodologie didattiche della programmazione in corsi introduttivi. L'analisi consente, inoltre, di scoprire inefficienze e porvi rimedio, oppure di identificare opportunità e nuove conoscenze che possono migliorare il processo di insegnamento della programmazione. Pur essendo il target di studenti molto specifico, l'analisi di questi dati può fornire informazioni molto interessanti e può essere in grado di individuare anche comportamenti inaspettati da parte degli studenti. I log, per loro natura, sono in grado di riportare qualsiasi azione che è stata effettuata da parte degli studenti all'interno dell'ambiente di sviluppo. Come anticipato nel capitolo precedente, Thonny registra qualsiasi comportamento dell'utente, nel contesto dell'ambiente di sviluppo, etichettando con un nome ciascuna azione effettuata.

4.2 Metodologia e progettazione

L'analisi dei log generati da Thonny è stata effettuata seguendo tutti gli elementi costitutivi di una pipeline di dati digitali: dall'acquisizione, pre-processamento ed estrazione di conoscenza mediante tecniche statistiche, fino alla visualizzazione dei risultati. Nello specifico il processo di analisi può essere riassunto nelle seguenti fasi:

1. Selezione e acquisizione dei dati: processo che racchiude la raccolta dei dati e lo studio degli scopi e degli obiettivi della ricerca. I dati dopo essere stati raccolti vengono modellati in forma tipicamente tabellare in modo da costruire un dataset da poter analizzare.

2. Pre-processamento dei dati: si tratta del processo che corrisponde alla pulizia dei dati o all'eliminazione del rumore. Questa fase è inoltre utile all'organizzazione preparativa dei dati da analizzare.

3. Trasformazione dei dati: dopo aver effettuato la fase di pre-processamento i dati vengono selezionati, unificati e consolidati in formati adatti all'analisi.

4. Estrazione di conoscenza: in questa fase vengono elaborati i dati mediante tecniche statistiche al fine di estrarre conoscenza e ritrovare pattern statistici nei dati.

5. Visualizzazione dei dati: la visualizzazione di dati copre tutto il processo di analisi. L'esplorazione visuale dei dati e la relativa rappresentazione grafica permette di identificare in tutte le fasi dell'analisi fenomeni e caratterizzazioni specifiche dei dati.

4.2.1 Strumenti e librerie utilizzate

L'analisi è stata interamente svolta con il linguaggio di programmazione Python. Python è un linguaggio che si adatta perfettamente all'analisi dei dati. Questo è dovuto al forte supporto e disponibilità di tutta una serie di librerie open source per scopi diversi, incluso il calcolo scientifico. Pertanto, non sorprende affatto che tale linguaggio si sia affermato nel campo della data science.
Per l'analisi sono stati utilizzati questi strumenti e librerie:

- Numpy: è una libreria open source per il linguaggio di programmazione Python, che aggiunge supporto a grandi matrici e array multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati.

- Pandas: è una libreria software per la manipolazione e l'analisi dei dati. In particolare, offre strutture dati e operazioni per manipolare tabelle numeriche e serie temporali.

- Jupyter Notebook: è un'applicazione web open source che consente di includere testo, video, audio e immagini e offre la possibilità di eseguire codice in di-

versi linguaggi di programmazione. Jupyter Notebook è particolarmente adatto per l'analisi dei dati esplorativi, e per mostrare facilmente i risultati dell'analisi effettuata.

- Matplotlib: è una libreria per la creazione di grafici per il linguaggio di programmazione Python e la libreria matematica NumPy. La libreria matplotlib consente di disegnare grafici 2D in vari modi.

- Seaborn: è una libreria di visualizzazione dati Python basata su matplotlib. Fornisce un'interfaccia di alto livello per disegnare grafici statistici graficamente attraenti e informativi.

4.3 Implementazione

Di seguito verranno descritti in modo dettagliato tutti i processi di implementazione per la realizzazione dell'analisi dei log.

4.3.1 Acquisizione dei dati e creazione dataset

I dati sono stati acquisiti durante le lezioni e sono stati organizzati in cartelle numerate secondo l'identificativo anonimo degli studenti. Ad ogni lezione di laboratorio, negli ultimi minuti veniva chiesto agli studenti di fornire i log generati. I log sono stati raccolti in forma anonima, non è infatti possibile risalire allo specifico studente analizzandoli. Sono stati inoltre raccolti i voti dell'esame finale di ciascuno studente che ha acconsentito all'attività di ricerca. I dati sono stati raccolti dai docenti del corso, i quali hanno provveduto a fornirli, anonimizzati, per lo sviluppo di questa tesi.

4.3.2 Lettura esplorativa e comprensione dei log

Prima di procedere alla creazione del dataset è stato implementato uno script che fosse capace di leggere i log in maniera "intelligente". Esportare i log dell'ambiente di sviluppo Thonny utilizzato durante tutto il corso è molto semplice. Thonny esporta i log in formato .txt ed assegna al file di testo la data e l'ora in cui l'utente ha aperto

l'ambiente di sviluppo. Tuttavia, anche se il nome del file si riferisce alla data di esportazione, il contenuto si riferisce all'intera giornata di laboratorio. Come accennato nel capitolo precedente, potrebbe però accadere di trovare nei log eventi non appartenenti al momento esatto del laboratorio in quanto gli alunni possono aver utilizzato Thonny anche qualche ora prima (o dopo) la lezione vera e propria senza poi chiuderlo e riaprirlo all'inizio (o alla fine) della lezione.
I log sono stati raggruppati in cartelle numerate, ogni cartella contiene tutti i log appartenenti ad uno specifico studente.Tuttavia ogni cartella può contenere un numero diverso di file di testo. Può capitare infatti che uno studente fosse assente, o abbia per errore esportato log di giorni diversi da quello del laboratorio, o, più semplicemente, che abbia aperto e chiuso Thonny più volte, facendo partire ogni volta la generazione di un nuovo file. Tra tutti i log, infatti, non è raro trovare studenti con pochissimi file di testo di log e altri con molti file di log. Questo è fattore da prendere in considerazione, in quanto potrebbe avere una grande rilevanza sull'analisi. Per questo motivo il processo di acquisizione dei log è fondamentale per la realizzazione di un buon dataset. Ogni file di testo è organizzato secondo un array json. Ogni elemento dell'array del singolo file di log contiene una moltitudine di oggetti json, ogni oggetto a sua volta contiene i valori corrispondenti al singolo evento ad un certo tempo t. Ogni oggetto dell'array, dunque, può contenere diversi valori. Thonny non memorizza molte variabili, in totale infatti esse sono poco più di 20. Essendo dunque un numero ragionevole di valori possibili, si è deciso di inglobarli tutti nel dataset. Se da un lato pochi valori permettono di creare un dataset con poche features, rendendo il dataset più leggibile e comprensibile ad occhio umano, dall'altro un dataset con poche features potrebbe fornire meno informazioni. Dalla pagina ufficiale di github della repository di Thonny, non sono illustrati nello specifico tutti i valori che possono assumere i log. Nella pagina relativa alla documentazione sui log, gli autori dichiarano che i valori dei log prendono spunto da Tkinter. Tkinter è un toolkit GUI utilizzato in python per creare GUI user-friendly. Tkinter è il framework GUI più comunemente usato e più semplice disponibile in python. Tkinter utilizza un approccio orientato agli oggetti per creare GUI.

4.3.3 Script per la creazione del dataset iniziale

Dopo aver studiato e visionato la struttura dei log di Thonny, si è proceduto con la creazione del dataset iniziale. Si è deciso di creare uno script in Python che fosse in grado di leggere tutti i dati e accorparli in un unico dataset seguendo alcune regole che di seguito verranno illustrate. Come anticipato nel paragrafo precedente, ogni oggetto dell'array json contiene diversi possibili valori. Si è deciso dunque di accorpare tutti i possibili valori nel dataset. In questo modo ogni evento corrisponderà ad una riga del dataset. Le colonne saranno rappresentate da tutte le possibili features che i log hanno generato. In questo modo saranno sicuramente presenti molti valori NaN che verranno gestiti nella fase di pre-processamento dei dati.
Il primo problema riscontrato durante questa fase è stato quello di creare uno script che fosse in grado di leggere ciclicamente in maniera temporale tutti i log. La prima difficoltà riscontrata è stata, infatti, quella di leggere i dati in maniera temporalmente corretta. Per ovviare a questo problema nella fase di accesso ai file di testo contenenti i log è stato necessario ordinare i file in sequenza temporale per ogni cartella relativa ad uno studente. Sono stati implementati, dunque, due cicli annidati. Il primo ciclo si occupa di accedere alle cartelle mentre il ciclo annidato si occupa di ciclare tra i file di testo dei log. In entrambi i cicli è stato necessario ordinare le cartelle in maniera crescente. In questo modo lo script parte dal primo studente fino ad arrivare all'ultimo in maniera crescente. Oltre alla features presenti nei log, ne sono state aggiunte altre per definire il dataset in maniera più rappresentativa. Una di queste è la features che corrisponde all'identificativo dello studente. Infatti per ogni istanza presente nel dataset sarà presente anche una feature che definisce l'identificativo dello studente. Per attuare ciò, dato che ogni cartella si riferisce all'identificativo dello studente, all'interno del ciclo è stato memorizzato nella variabile studentID l'id dello studente. Oltre ai due cicli annidati è stato inserito un terzo ciclo annidato che si occupa di scorrere tutti gli oggetti dell'array di ciascun file.
Ogni oggetto appartenente all'array del log corrispondente è stato memorizzato all'interno di una lista. Sono state utilizzate le liste anziché creare un dataframe pandas in quanto per questo tipo di calcoli iniziali le liste sono computazionalmente più efficienti. In questa fase di creazione del dataset sono infatti frequenti cicli. Effettuare delle itera-

zioni è infatti più efficiente su liste Python rispetto ad un dataframe Pandas. Nel ciclo più interno infatti si scorre ogni elemento dell'array che viene a sua volta modificato e memorizzato in una lista. Dopo aver effettuato le modifiche necessarie alla creazione della lista di eventi generati dai log, la lista viene estesa aggiungendo di volta in volta i dati relativi ai log generati per ogni studente. In questo modo sia la lettura che la scrittura degli eventi è più efficiente e veloce. In questa fase è stata effettuata anche un'operazione preliminare di pulizia di dati. In particolare si è deciso di aggiungere una feature che definisce in maniera chiara il tipo di errore che è stato evidenziato dall'ambiente di sviluppo durante il laboratorio. Thonny, infatti, registra nei log qualsiasi messaggio di errore evidenziato all'utente dopo un run. Tuttavia, spesso nei log sono presenti messaggi di errore molto lunghi e poco utili ad un'analisi specifica sugli errori. Dato che una parte rilevante di questa analisi è la fase di analisi degli errori, nel ciclo più interno nel quale si scorre ogni elemento all'interno di ogni oggetto dell'array JSON, è stata applicata una funzione in grado di estrarre solo il tipo di errore evidenziato. In questo modo la fase di analisi degli errori può essere più specifica e categorizzata.
Dopo aver implementato la lista contenente tutti gli eventi di tutti gli studenti in maniera ordinata temporalmente, si è proceduto con la creazione del dataset vero e proprio per l'inizio dell'analisi. Il dataset è un dataframe Pandas, nel quale le righe corrispondono agli eventi dei log e le colonne corrispondono alle variabili (o features) descritte in precedenza.

4.3.4 Il dataset

Il dataset creato è composto da 2109852 istanze, che corrispondono agli eventi generati da Thonny per tutti gli studenti che hanno partecipato alla raccolta dei log. In totale le features sono 29:

- 23 features sono di tipo object.
- 5 features sono di tipo float64.
- 1 feature è di tipo int64.

Nell' immagine 4.1 è possibile notare nel dettaglio il data type di ogni feature presente nel dataset:

```
Data columns (total 29 columns):
 #   Column                Dtype
---  ------                -----
 0   view_id               object
 1   view_class            object
 2   sequence              object
 3   time                  object
 4   student_ID            int64
 5   score                 object
 6   editor_id             float64
 7   editor_class          object
 8   text_widget_id        float64
 9   text_widget_class     object
 10  widget_id             float64
 11  widget_class          object
 12  index                 float64
 13  text                  object
 14  tags                  object
 15  trivial_for_coloring  object
 16  trivial_for_parens    object
 17  text_widget_context   object
 18  index1                float64
 19  index2                object
 20  error_type            object
 21  filename              object
 22  save_copy             object
 23  cmd_line              object
 24  command_text          object
 25  input_text            object
 26  widget                object
 27  command_id            object
 28  denied                object
dtypes: float64(5), int64(1), object(23)
```

Figura 4.1: Features del dataset.

Successivamente, per effettuare le analisi sulla base del tempo, la feature time che corrisponde al timestamp relativo all'evento è stata convertita nel tipo datetime di pandas.

4.3.5 Gestione delle sequenze temporali

Come anticipato nei capitoli precedenti, Thonny genera log basati sul timestamp di ciascuna azione. L'analisi dei log verte principalmente sulle lezioni di laboratorio e non sugli esercizi che gli studenti hanno effettuato a casa o al di fuori del contesto del

corso. Nonostante sia stato esplicitamente chiesto agli studenti di fornire i log generati da Thonny solo nell'intervallo di tempo che corrispondeva alle ore di laboratorio effettuate,all'esplorazione dei dati è emersa qualche azione registrata anche in giornate o ore differenti dalle lezione effettive di laboratorio. Si è deciso di considerare solo i log generati durante i laboratori in quanto, oltre a renderli facilmente confrontabili mettendoli tutti sullo stesso piano, sono sicuramente più affidabili e appartenenti allo specifico argomento di laboratorio della giornata. Infatti, non è inusuale che uno studente svolga esercizi di argomenti precedenti anche giorni dopo la lezione. Inoltre, data la complessità della gestione di dati temporali si è deciso di scartare ogni altro evento che non appartenesse alle giornate di laboratorio. Durante la fase di data cleaning, dunque, si è proceduto con l'eliminazione dei dati non appartenenti alle lezioni di laboratorio.

4.3.6 Feature target

La feature target del dataset corrisponde al voto finale dello studente. Oltre ai log sono stati anche raccolti i voti dell'esame degli studenti in forma anonima. Nello script utilizzato per la creazione del dataset è stata creata una funzione che è in grado di leggere i voti degli studenti ed associare il rispettivo voto a ciascun log. Alla feature target sono stati associati diversi valori. Per gli studenti che non hanno ancora sostenuto l'esame o non l'hanno passato con una votazione di almeno 18, il target per quello studente assume il valore di "fail". In generale le categorie scelte da associare al target per le votazioni degli studenti possono essere riassunte in questo modo:

- **Failed**: corrisponde ad un esito negativo dell'esame (votazione minore di 18) oppure ad un esame non ancora sostenuto.
- **Passed**: corrisponde ad un esame sostenuto e passato con un voto compreso tra 18 e 26 compreso.
- **Excellent**: corrisponde ad un esame sostenuto e passato con votazione maggiore di 26.

4.3.7 Gestione dei valori NaN

Dopo aver costruito il dataset si è proceduto alla fase di data cleaning. Il primo passaggio è stato quello di gestire i valori mancanti. I valori mancanti o anche definiti valori NaN (not a number) presenti nel dataset erano molti. Si è utilizzata la libreria Missingno per visualizzare la distribuzione dei valori NaN. Missingno è una libreria Python e compatibile con Pandas. Dato che ogni oggetto della lista del file di log contiene solo le informazioni necessarie per identificare uno specifico evento, è inevitabile che il dataset presentasse molti valori mancanti. Tuttavia, essendo le possibili features relativamente poche, si è gestito il problema dei valori NaN in maniera efficace e valida per il tipo di analisi da effettuare. Utilizzando la libreria Missngno, infatti, è stato possibile visualizzare per ogni features il numero di valori mancanti. La figura 4.2 mostra la rappresentazione grafica dei valori NaN effettuata mediante tale libreria.

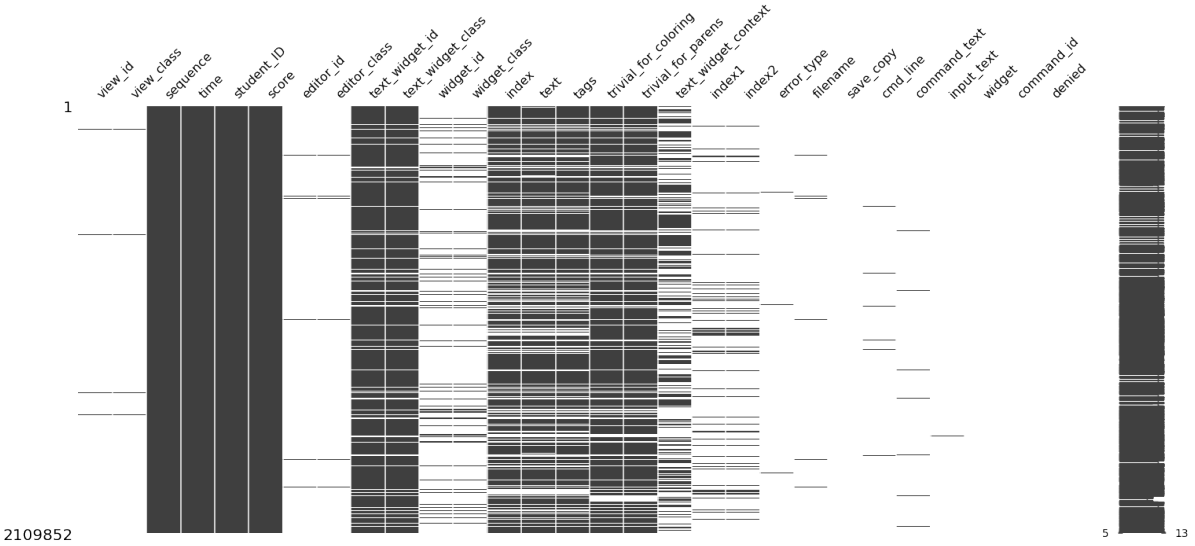

Figura 4.2: Rappresentazione grafica dei valori mancati nel dataset, la parti in bianco evidenziano i valori mancanti.

Per avere maggiori informazioni è stata creata una funzione che preso in input il dataframe è in grado di calcolare la percentuale di valori mancanti. Nel dataframe, come è possibile notare anche dalla figura precedente, sono presenti feature che non

possiedono valori NaN. Questo non suscita alcun dubbio, in quanto le features "time" e "sequence" sono le uniche presenti in tutte gli oggetti delle liste dei file di testo di log. Infatti la feature "time" si riferisce al timestamp dell'evento, e la feature "sequence" si riferisce al nome dell'evento stesso. Al contrario sono evidenti alcune features che hanno un'alta percentuale di valori NaN, mentre altre hanno la totalità di valori mancanti. Per queste ultime si è deciso di eliminarle. Queste features sono: "denied", "command_id" e "widget". Sono presenti anche features con percentuali di valori mancanti molto alte che superano in alcuni casi anche il 99%. Dato che il dataset possiede poche features nella sua totalità e dato che per questo tipo di analisi queste features non inficiano negativamente si è deciso di mantenerle.

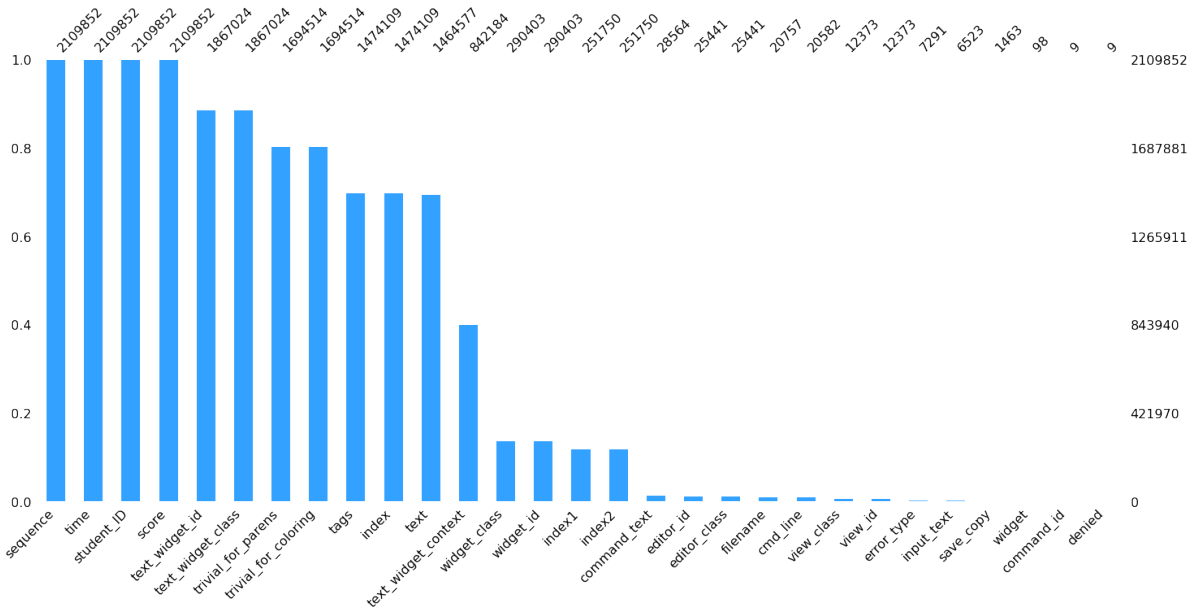

Figura 4.3: Istogramma relativo al numero di valori non nulli per ciascuna colonna presenti nel dataset in ordine decrescente.

4.3.8 Le informazioni rilevanti dei log: errori, run e codice incollato

Dopo aver costruito il dataset ed averlo trasformato in un padas dataframe, si è proceduto con l'analisi dei log mediante un Jupyter Notebook. In particolare, l'analisi si è concentrata sulle informazioni più interessanti e rilevanti per un'analisi di questo tipo. Dopo aver studiato i dati ed aver riflettuto sulle possibili interpretazioni e tecniche da applicare, è emerso che i dati più interessanti su cui fosse utile effettuare un'analisi approfondita sono:

- Gli errori e nello specifico il tipo di errore evidenziato nell'ambiente di sviluppo.
- I run e il numero di volte che gli studenti hanno eseguito il codice.
- Il codice incollato.

Inoltre questi dati possono essere interessanti soprattutto in una prospettiva temporale. Infatti, dato che ad ogni evento è associato un timestamp è possibile tracciare l'andamento di queste informazioni nel tempo. Questo può essere molto importante, in quanto è possibile estrapolare delle informazioni aggiuntive riguardo il progresso degli studenti nel tempo.

Errori

Tra le informazioni più interessanti per l'analisi che i log di Thonny forniscono, vi sono gli errori del codice eseguito dagli studenti durante le esercitazioni di laboratorio. Per la gestione degli errori, durante la fase di creazione del dataset nello script creato in Python, è stato aggiunto un elemento in ogni evento nel caso in cui nel campo "text" era presente la parola Error. Infatti ogni errore evidenziato dall'ambiente di sviluppo viene registrato sotto i campi "sequence" con il valore "TextInsert" e "text" con il valore "Error". Lo script è stato implementato secondo la logica per cui, una volta trovato l'evento errore, vengono presi solo i primi caratteri del testo in modo tale da selezionare unicamente il tipo di errore e non tutto il messaggio di errore. Questo è dovuto al fatto che il contenuto del campo "text" registra tutto il messaggio di errore. Ma come prima

parola viene mostrato il tipo di errore. Per separare il tipo di errore dal testo completo è stato utilizzato il metodo split che è in grado di dividere il testo in un array di stringhe. Come parametro è stato dato il carattere ":" che è quello che separa l'errore dal testo. Dopo questo passaggio lo script verifica che la stringa sia minore di 50 caratteri, per verificare la correttezza del tipo di errore. Infine, la stringa caratterizzata dal tipo di errore viene aggiunta al dizionario. In questo modo nel dataset che verrà trasformato in un dataframe sarà presente la feature "error_type" che per l'appunto conterrà il tipo di errore nel caso di eventi con errori.

Run

Un altro aspetto particolarmente interessante dei log generati da Thonny è l'informazione che riguarda le volte in cui gli studenti premono il tasto Run per eseguire il codice. All'evento in cui il campo "command_text" dei log generati presente la stringa "%Run", corrisponde all'azione dello studente in cui ha premuto run su Thonny per eseguire il codice scritto. Come per gli errori, è stato implementato un dataframe che contiene solo gli eventi relativi ai run.

Codice incollato

Nel campo "sequence" il valore "Paste" corrisponde al testo incollato. Anche in questo caso è stato creato un dataframe contenente tutti gli eventi relativi al testo incollato di tutti gli studenti che hanno partecipato alla raccolta dei log.

Le analisi sono state effettuate mediante le principali librerie e strumenti descritti nei capitoli precedenti. Nel capitolo successivo saranno illustrati i risultati.

Capitolo 5

Risultati e discussione

In questo capitolo si discute dei risultati dell'analisi dei log e si confrontano i risultati con quelli di altri studi analoghi presenti in letteratura.

5.1 I voti degli esami

Gli studenti che hanno partecipato allo studio fornendo i loro log in forma anonima hanno anche acconsentito ad accedere al proprio voto finale. Infatti, oltre ai log, sono stati raccolti i voti degli esami svolti dagli studenti. Ogni voto appartiene all'identificativo associato allo studente che ha fornito i log, in modo da mantenere l'anonimato. Come anticipato nel capitolo precedente, la votazione finale rappresenta il target del dataset. Si è deciso di suddividere i voti sulla base di tre categorie: failed (voto minore di 18 o esame non ancora sostenuto), passed (voto compreso tra 18 e 26), excellent (voto maggiore di 26).
Come raffigura l'immagine 5.1 le informazioni relative alla votazione finale possono essere riassunte in questo modo:

- Il numero di studenti che hanno sostenuto e passato l'esame con una votazione maggiore di 26 sono il 50%, poco più di 50.
- Gli studenti che hanno passato l'esame con un voto compreso tra 18 e 26 sono il 14.15%, poco più di 30.

- Gli studenti che non hanno sostenuto l'esame o hanno ricevuto un voto minore di 18 sono il 35.85%.

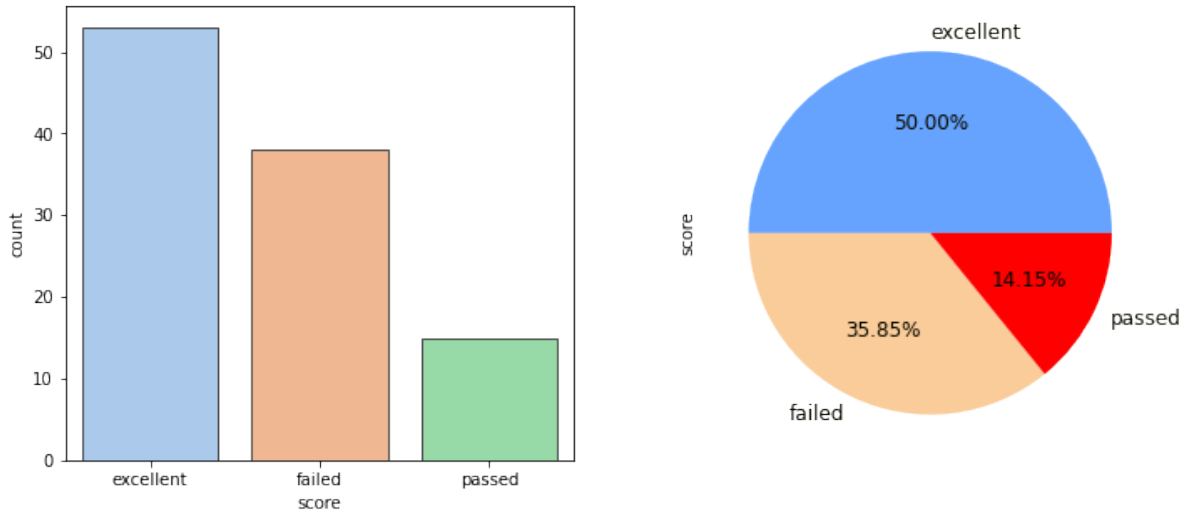

Figura 5.1: Numero di studenti sulla base del punteggio finale.

L'immagine successiva 5.2 raffigura, in maniera analoga all'immagine precedente le percentuali dei voti finali, ma in questo caso le percentuali si riferiscono a tutte le istanze del dataset. Dato che l'analisi tiene conto delle istanza che rappresentano il singolo evento di log degli studenti si è ritenuto necessario rappresentare anche la quantità del target sulla base di tutti gli eventi. Essendo il numero di eventi molto alti, la libreria utilizzata per i grafici Seaborn utilizza per il conteggio sull'asse delle y la notazione scientifica 1e6 che corrisponde a $1x10^6$, dunque 1 milione.
Tali informazioni possono essere riassunte in questo modo:

- Il numero di istanze, quindi di eventi dei log, appartenenti a studenti che hanno sostenuto e passato l'esame con una votazione maggiore di 26 sono il 63.91%, poco più di 1.3 milioni.

- Il numero di eventi dei log appartenenti a studenti che hanno passato l'esame con un voto compreso tra 18 e 26 sono il 13.14%, poco più di 400 mila.

- Il numero di eventi dei log appartenenti a studenti che non hanno sostenuto l'esame o hanno ricevuto un voto minore di 18 sono il 22.94%, poco più di 200 mila

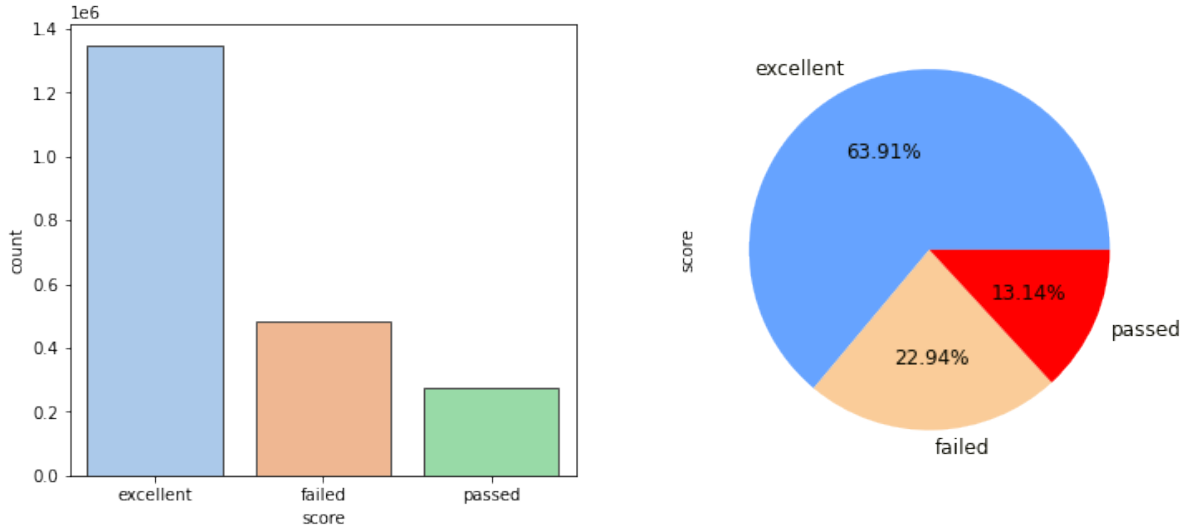

Figura 5.2: Numero di eventi dei log appartenenti agli studenti sulla base del punteggio finale.

Le due immagini sono molto rappresentative e ci forniscono delle importanti informazioni sui log e sul target. Infatti se da un lato i dati mostrano che la metà degli studenti ha passato l'esame con voti superiori al 26, dall'altro è di grande rilievo notare che tra tutti gli eventi dei log presenti nei dataset il 64% circa appartengono a questi studenti. Questo dato ha un grosso peso per l'analisi. Infatti, dato che l'analisi avviene sugli eventi dei log, è importante ricordare l'intera analisi avviene sulla base del secondo grafico. Se il primo grafico, infatti, ci fornisce una panoramica del target relativo agli studenti e al loro voto finale, il secondo grafico ci fornisce le informazioni relative a tutti gli eventi dei log registrati e inseriti nel dataset. Il primo grafico, dunque, definisce semplicemente sulla base del campione di studenti quanti di questi hanno passato l'esame con voti maggiori di 26, quanti l'hanno passato con un voto compreso tra 18 e 26, ed infine quanti studenti non hanno svolto l'esame o sono stati bocciati. Inoltre, queste informazioni ci forniscono ulteriori spunti interessanti per l'analisi. Dato che il numero

di studenti che hanno passato l'esame con voti alti sono la metà di tutti gli studenti che hanno partecipato all'analisi, è interessante notare come tra tutti gli eventi presenti nel dataset la percentuale relativi a questi studenti sia più alta.
La prima interessante informazione che possiamo ricavare da quanto detto e mostrato è che gli studenti che hanno avuto voti più alti negli esami, sono gli studenti che hanno generato molti più eventi nell'ambiente di sviluppo Thonny e dunque sono stati quelli che si sono esercitati di più e molto probabilmente hanno scritto più codice degli altri. Inoltre è interessante notare che se da un lato la percentuale di studenti che hanno passato l'esame con un voto compreso tra 18 e 26 sia molto simile alla percentuale di eventi presenti nel dataset relativi a questi stessi studenti, la percentuale di studenti che non ha passato l'esame o non l'ha ancora svolto è nettamente minore rispetto alla percentuale di eventi presenti nei log degli stessi studenti. In maniera del tutto opposta a quanto detto per gli studenti che hanno ottenuto voti maggiori, possiamo affermare che gli studenti che hanno ottenuto una bocciatura all'esame o non ha lo hanno ancora effettuato, sono quelli che hanno molto probabilmente svolto meno esercitazioni e scritto meno codice rispetto alle altre due categorie di studenti.

5.2 Errori

Come discusso nel capitolo precedente, analizzare gli errori più frequenti degli studenti durante i laboratori di programmazione può fornire informazioni rilevanti per valutare la comprensione effettiva degli argomenti e la progressione nel tempo per quanto riguarda le abilità nella programmazione. In questo caso sono stati analizzati nello specifico, tra tutti gli eventi presenti nei log del dataset, solo quelli che appartenevano ad un evento corrispondente ad un errore, ovvero quelli la cui feature error_type conteneva il tipo di errore evidenziato dall'ambiente di sviluppo dopo l'esecuzione di codice scritto dagli studenti. Da una prima analisi generale è emerso che:

- Il dataset contiene 7291 eventi che corrispondono ad errori.
- La media del numero degli errori per studente è di 78.20.

L'immagine 5.3 raffigura la frequenza dei tipi di errori più comuni tra gli studenti durante le ore di laboratorio.

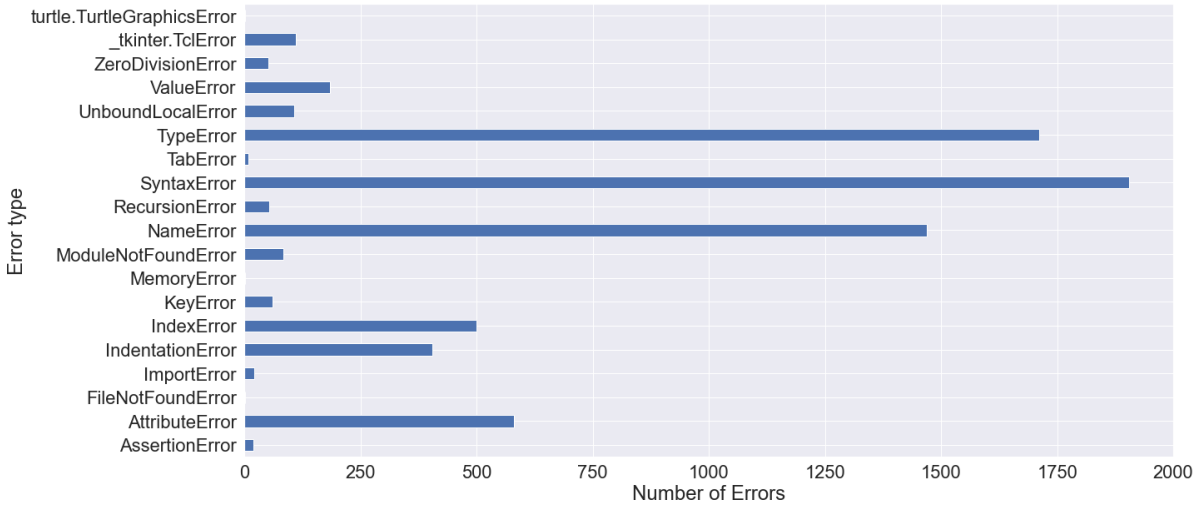

Figura 5.3: Numero degli errori per tipo di errore.

In python si possono distinguere due tipi di errori: gli errori di sintassi e le eccezioni. Gli errori di sintassi sono errori che il compilatore riconosce sulla base della sintassi scorretta. Anche se un'istruzione, o un'espressione, è sintatticamente corretta, può causare un errore quando si tenta di eseguirla. Gli errori rilevati durante l'esecuzione sono chiamati eccezioni. L'ultima riga del messaggio di errore indica cos'è successo. La stringa stampa quale tipo di eccezione è occorsa ed il nome dell'eccezione stessa. Ciò è vero per tutte le eccezioni built-in, ma non è necessario che lo sia per le eccezioni definite dall'utente. I nomi delle eccezioni standard sono identificatori built-in, non parole chiave riservate.
Tra tutti i tipi di errori presenti nel dataset principalmente tre hanno una frequenza maggiore rispetto agli altri. Questi sono: TypeError, SyntaxError, NameError. Nello specifico queste tre tipologie di errori si riferiscono principalmente a queste caratteristiche:

- SyntaxError: Sono errori di sintassi, noti anche come errori di parsing, che si riferiscono per l'appunto ad errori che riguardano la sintassi del codice. L'analizzatore

sintattico ('parser') riporta la riga incriminata e mostra una piccola freccia che punta al primo punto in cui è stato rilevato l'errore. L'errore è causato dal token che precede la freccia. Nel messaggio di errore mostrato dall'ambiente di sviluppo vengono stampati il nome del file e il numero di riga, in modo che si sappia dove andare a guardare. Questo tipo di errore è presente in più di 1900 eventi.

- TypeError: Un tipo di eccezione che viene sollevata quando un'operazione o una funzione viene applicata ad un oggetto di tipo inappropriato. Il valore associato è una stringa contenente i dettagli sul tipo errato. Questo tipo di errore è presente in più di 1700 eventi.

- NameError: Un tipo di eccezione che viene sollevata quando un nome locale o globale non viene trovato. Questo si applica solo ai nomi non qualificati. Il valore associato è un messaggio di errore che include il nome che non può essere trovato. Questo tipo di errore è presente in circa 1500 eventi.

Altri tipi di errori, meno frequenti in confronto ai tre errori citati sopra, ma rilevanti per l'analisi sono:

- IndexError: Un tipo di eccezione che viene sollevata quando si cerca di accedere ad un indice che è invalido. Questo tipo di errore è presente in circa 500 eventi.

- IdentationError: Errore che fa parte della classe di errori di tipo sintattici e si riferisce ad un errata indentazione del codice. Questo tipo di errore è presente in circa 450 eventi.

- AttributeError: Un tipo di eccezione che viene sollevata quando un riferimento ad un attributo o ad un assegnamento fallisce. (Nel caso in cui un oggetto non supporti affatto i riferimenti ad un attributo o gli assegnamenti ad un attributo, viene sollevata un'eccezione di tipo TypeError). Questo tipo di errore è presente in circa 600 eventi.

- ValueError: Un tipo di eccezione che viene sollevata quando un'operazione o funzione built-in ricevono un argomento che ha il tipo giusto ma valore inappropriato,

e la situazione non viene descritta da una eccezione più precisa, come IndexError. Questo tipo di errore è presente in circa 200 eventi.

La figura 5.4 sintetizza chiaramente quanto detto fino ad ora mostrando i 7 errori più comuni e la loro quantità nel dataset.

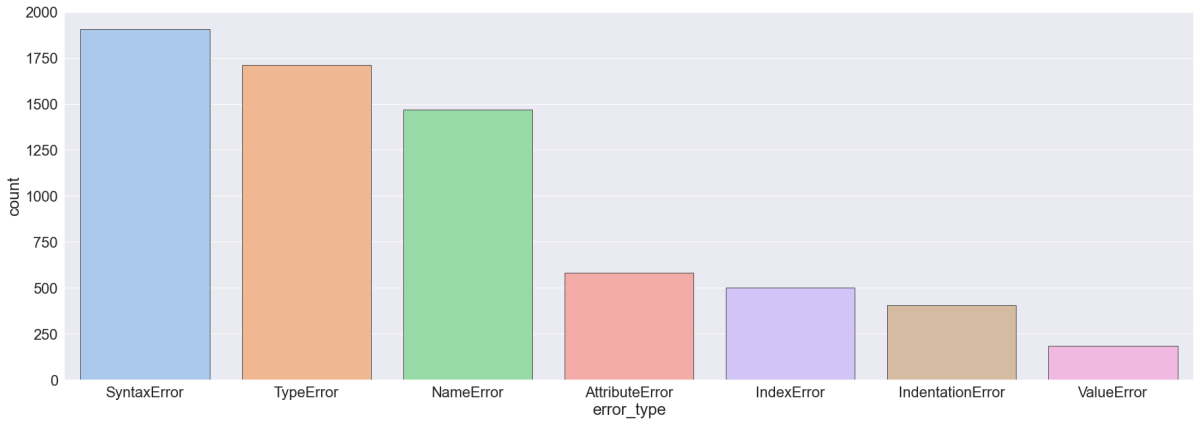

Figura 5.4: Numero dei 7 errori più frequenti.

5.2.1 Analisi degli errori in letteratura

I tipi di errori comuni nella programmazione che i programmatori inesperti commettono e che si scontrano cercando di volta in volta di risolvere durante il processo di apprendimento, sono stati a lungo di interesse per i ricercatori. Queste informazioni, tuttavia, non hanno una correlazione diretta con i tipi di errori che gli studenti commettono, a causa dell'inesattezza e dell'imprecisione dei tipi di errori. McCall e Kölling [42] suggeriscono nello studio degli errori di non limitarsi solo alla frequenza ma anche ad un ordinamento che tenga conto della difficoltà di correzione gli errori, diverso da una classificazione per sola frequenza. In maniera analoga ai risultati mostrati nel paragrafo precedente, anche nell'analisi di McCall e Kölling è emerso che il maggior numero di errori è di tipo sintattico. I risultati dello studio effettuato da McCall e Kölling, inoltre, indicano che una quantità significativa di errori più gravi è difficile da classificare sulla base della difficoltà di correzione con precisione.

Un lavoro simile all'analisi degli errori di questa tesi è stato svolto da Smith e Rixner [43], che presentano un'analisi quantitativa di un ampio set di dati di programmi scritti nel linguaggio Python da studenti novizi. L'analisi dipinge un quadro sfaccettato degli errori che gli studenti incontrano, fornendo informazioni sulla distribuzione, la durata e l'evoluzione di questi errori. Dallo studio emergono alcuni pattern che indicano che alcune aree meritano un'attenzione aggiuntiva da parte di istruttori e ricercatori. I risultati mostrano, ad esempio, che i TypeErrors non erano solo uno degli errori di runtime più frequenti, ma era anche più probabile che si ripresentassero più frequentemente negli esercizi successivi, ed erano uno dei più probabili a persistere anche nella versione finale dell'esercizio. Altri tipi di errori rilevati dall'analisi si verificavano meno frequentemente, ma persistevano più a lungo quando venivano visualizzati, come IndexErrors e KeyErrors. Secondo gli autori, per tali errori, gli istruttori farebbero bene a istruire gli studenti non solo su come evitarli, ma anche su pratiche di debug efficaci. Inoltre, in modo analogo ai risultati mostrati nel paragrafo precedente, i risultati dell'analisi di Smith e Rixner mostrano che gli errori di sintassi erano quelli più frequenti, ma concentrati in modo sproporzionato in un sottoinsieme più piccolo di studenti. Per rimediare a questo squilibrio gli autori suggeriscono che gli istruttori potrebbero dover identificare quegli studenti che hanno problemi con la sintassi al fine di fornire loro delle istruzioni mirate in modo da correggere eventuali errori comuni.

5.2.2 Difficoltà nella comprensione dell'errore

Alcuni studiosi sostengono che i messaggi di errore possono essere difficili da comprendere per i principianti, ostacolando il progresso e portando lo studente alla frustrazione. In risposta a tale problema, i ricercatori hanno esplorato vari approcci per migliorare i messaggi di errore, ma i risultati di questo filone attivo di ricerca sono attualmente contrastanti. Il confronto diretto dei risultati tra gli studi è impegnativo poiché questi in genere esaminano diversi tipi di miglioramenti dei messaggi di errore e riportano i risultati utilizzando metriche diverse. Denny, James e Becker [44] hanno osservato gli effetti del presentare a studenti novizi messaggi di errore del compilatore progettati utilizzando messaggi leggibili più facilmente, brevi e contenenti suggerimenti per la risoluzione. Gli

autori sostengono che tra le varie linee guida per la progettazione di messaggi di errore, 4 sono quelle più rilevanti:

- Aumentare la leggibilità dell'errore: I messaggi di errore non sono tipicamente progettati per essere "facili da leggere", ma piuttosto si presentano come spiegazioni brevi e concise per programmatori esperti. Questo è uno dei motivi principali per il quale gli studenti novizi hanno difficoltà ad apprendere e ad individuare l'errore effettuato nel codice.

- Ridurre il carico cognitivo: Sebbene vi siano stati alcuni tentativi di migliorare i messaggi di errore hanno utilizzando un feedback più dettagliato, come ad esempio mostrare sia il messaggio di errore originale insieme al messaggio migliorato, può essere controproducente in quanto può aumentare il carico cognitivo e quindi non essere più utile del messaggio originale.

- Utilizzare un tono positivo: Per gli studenti novizi può essere di grande aiuto messaggi di errore "educati" e "incoraggianti".

- Mostrare soluzioni o suggerimenti: Mostrare agli utenti soluzioni o suggerimenti può avere dei benefici durante il processo di apprendimento nella programmazione.

Per determinare con precisione il tempo e lo sforzo necessari per leggere e provvedere alla correzione degli esercizi dopo aver letto i messaggi di errore, James e Becker hanno provveduto a creare un'attività di debug in cui a tutti gli studenti veniva presentato lo stesso codice e quindi venivano riscontrati gli stessi errori. Seguendo, dunque, le linee guida dei messaggi di errore di programmazione hanno effettuato uno studio empirico sull'efficacia e l'utilità percepita del miglioramento dei messaggi di errore mostrando che quando i partecipanti incontravano messaggi migliorati, facevano meno invii e impiegavano meno tempo per completare un'attività di debug rispetto a quando i partecipanti incontravano i messaggi di errore standard. Dai questionari è emerso che quasi tutti gli studenti hanno riferito di aver letto i messaggi trovando, in modo schiacciante, i nuovi messaggi più utili per il loro lavoro rispetto ai messaggi di errore del compilatore originale. Gli studenti, inoltre, hanno sostenuto che i messaggi avanzati hanno ridotto in modo significativo il tempo e lo sforzo di debug.

Lo studio analizzato evidenzia, dunque, che il messaggio di errore standard può essere molto difficile da comprendere per i principianti, ostacolando in maniera evidente il progresso nell'apprendimento della programmazione. Un interessante soluzione a questo problema, nel caso in cui si decidesse di utilizzare Thonny come ambiente di sviluppo per insegnare la programmazione, potrebbe essere quella di incentivare gli studenti ad utilizzare plug-in esterni che aiutano nella comprensione dell'errore fornendo agli studenti messaggi più chiari ed esaustivi.

5.3 Analisi degli errori

Dopo la fase di data cleaning per quanto concerne gli errori, discussa nel capitolo 4, sono stati analizzati gli errori nel tempo sulla base del timestamp relativo ad ogni evento dei log. Se analizzati in corrispondenza dei mesi di laboratorio è possibile notare nella figuta 5.5 come il mese di aprile sia quello con il numero di errori maggiore, quasi 3000 errori. Il mese successivo, quindi il mese di maggio, è quello con il numero minore di errori, poco meno di 2000. Infine il mese di marzo è caratterizzato da poco meno di 2500 errori. Nello specifico, è bene ricordare che il mese di maggio è stato il mese in cui il numero di laboratori è stato minore rispetto agli altri. Infatti, a maggio sono stati svolti solo 3 laboratori. Questo potrebbe essere uno dei motivi per il quale in questo mese si sono registrati meno errori. Tuttavia, non è possibile dire lo stesso nel momento in cui si procede a comparare i mesi di marzo e aprile. Infatti, in entrambi i mesi sono stati effettuati gli stessi numeri di laboratori. Il numero di laboratori per il mese di marzo e aprile sono stati 4.

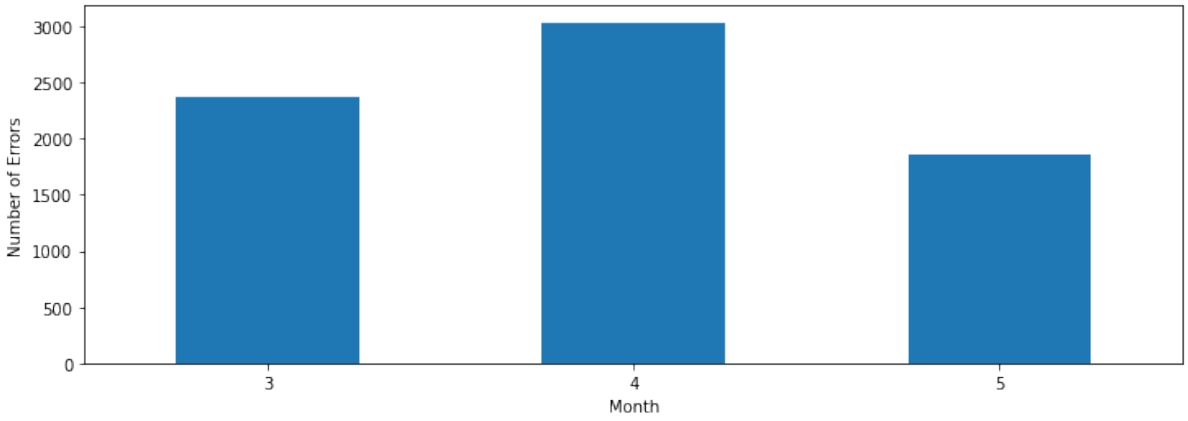

Figura 5.5: Numero degli errori nei mesi.

Prima di proseguire con l'analisi degli errori nel tempo più nel dettaglio, è bene ricordare che gli argomenti trattati nel primo mese sono stati quelli che, generalmente, risultano essere più semplici e di facile comprensione per gli studenti. Al contrario gli argomenti trattati nel secondo mese sono stati argomenti più complessi rispetto al primo mese. Nel mese di aprile, infatti, sono stati trattati argomenti come la ricorsione, dizionari e list comprehensions che sono generalmente, come discusso nei primi capitoli di questo elaborato, concetti di difficile comprensione per gli studenti che imparano a programmare per la prima volta. Dunque, è molto probabile che la quantità di errori maggiore nel secondo mese sia dovuta principalmente alla complessità degli argomenti trattati.

5.3.1 Analisi degli errori nel tempo: settimane e mesi

Analizzare gli errori nell'arco temporale delle settimane o dei mesi può fornire informazioni molto importanti sull'effettiva comprensione degli argomenti da parte degli studenti. La figura 5.6 fornisce una panoramica generale e più dettagliata della quantità di errori nelle settimane. La settimana 13 che corrisponde alla prima settimana in cui sono stati raccolti i log ovvero la terza settimana di marzo, si sono registrati un alto numero di errori, più di 1750. In maniera analoga anche la settimana 18 ha registrato un alto numero di errori. Questo è dovuto al fatto che durante queste settimane si sono

svolte più lezioni. Nel paragrafo successivo si andranno ad analizzare nello specifico gli argomenti trattati durante queste settimane e durante i giorni di queste settimane per individuare eventuali esercitazioni risultate complesse per gli studenti.

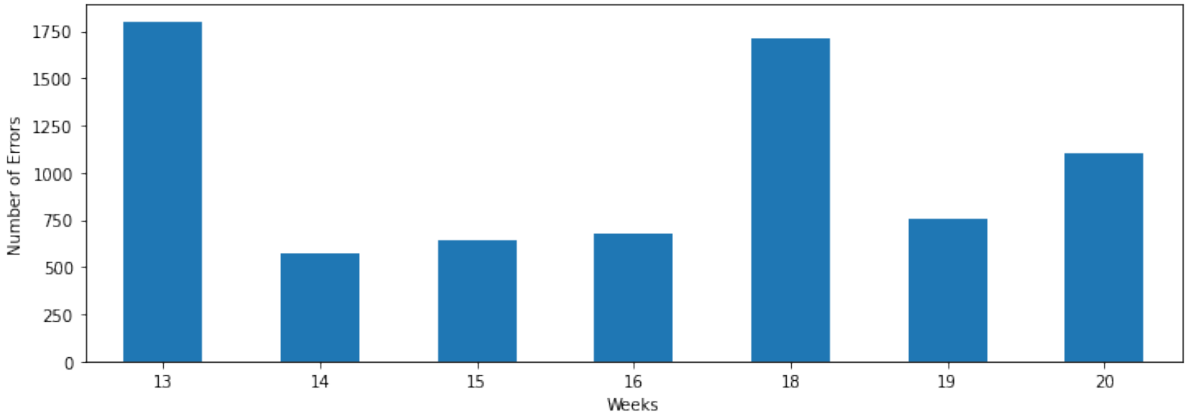

Figura 5.6: Numero di errori nelle settimane.

Gli argomenti dei laboratori relativi alle giornate sono consultabili nella tabella 3.3. Approfondendo nel dettaglio gli errori durante le giornate nei mesi di laboratorio è possibile notare come nel mese di marzo sia il giorno 26 quello con il maggior numero di errori, circa 1000, come rappresentato nella figura 5.7. Il giorno 23 sono stati registrati circa 800 errori e il giorno 30 poco meno di 600.

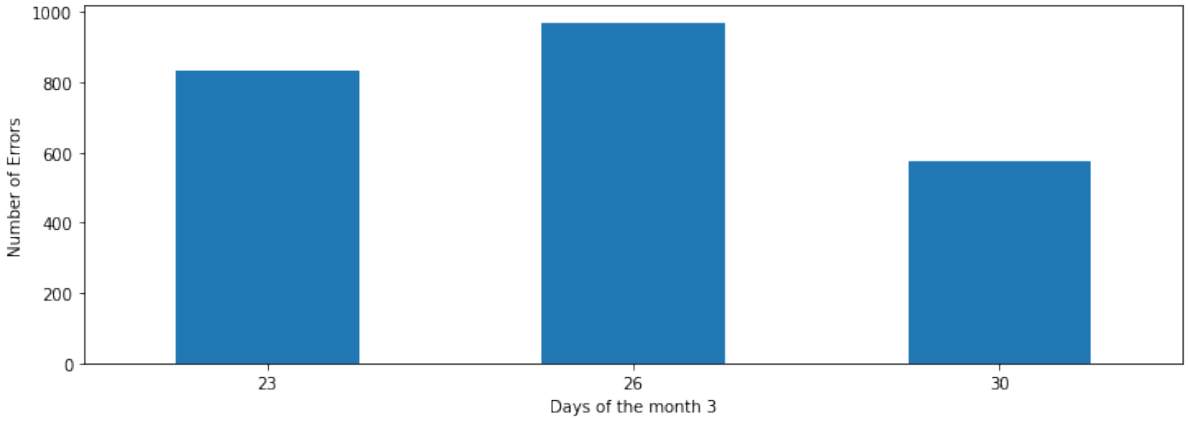

Figura 5.7: Numero di errori durante il mese di marzo nelle giornate di laboratorio del 23 (Iterazione determinata), 26 (Debugging), 30 (Iterazione indeterminata).

Per quanto riguarda il mese di aprile, come è possibile notare nella figura 5.8, il giorno in cui sono stati registrati più errori è stato il 30, con poco meno di 1200 errori. Mentre le giornate del 6,16 e 27 sono stati registrati interno ai 600 errori.

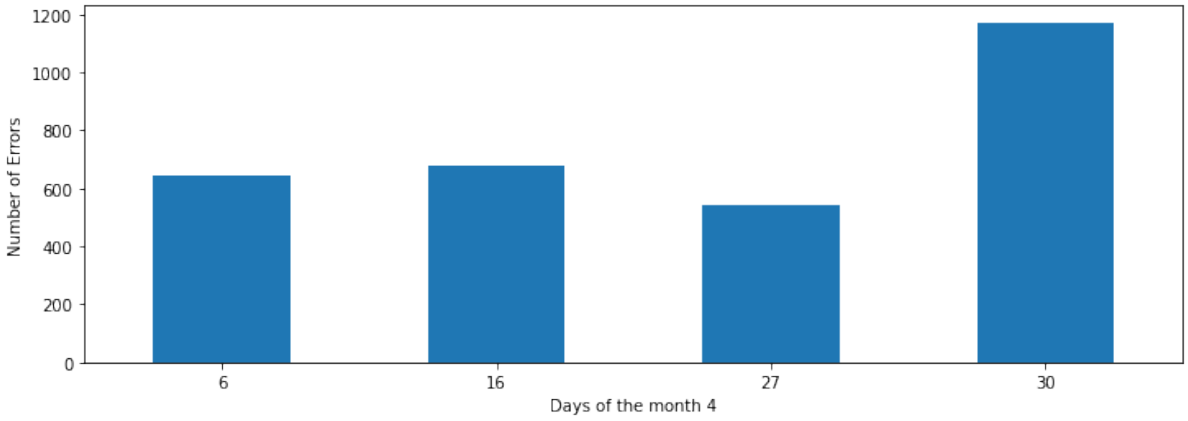

Figura 5.8: Numero di errori durante il mese di aprile nelle giornate di laboratorio del 6 (Liste), 16 (Ricorsione), 27 (Dizionari), 30 (Comprehension).

Infine l'ultimo mese, quello di maggio come riassunto nella figura 5.9, il giorno in cui sono state registrati più errori è stato il giorno 7 con poco più di 700 errori. Mentre

le giornate dell'11 e del 14 hanno registrato un numero di errori intorno a 600. Nei seguenti grafici mancano dati relativi al laboratorio 4, in quanto a seguito della fase di data cleaning i dati rimanenti erano molto pochi e irrilevanti, si è quinid deciso di non analizzarli. Nel paragrafo successivo si andranno ad analizzare nel dettaglio i tipi di errori relativi alle specifiche giornate considerando, in particolare, il tipo di argomento trattato.

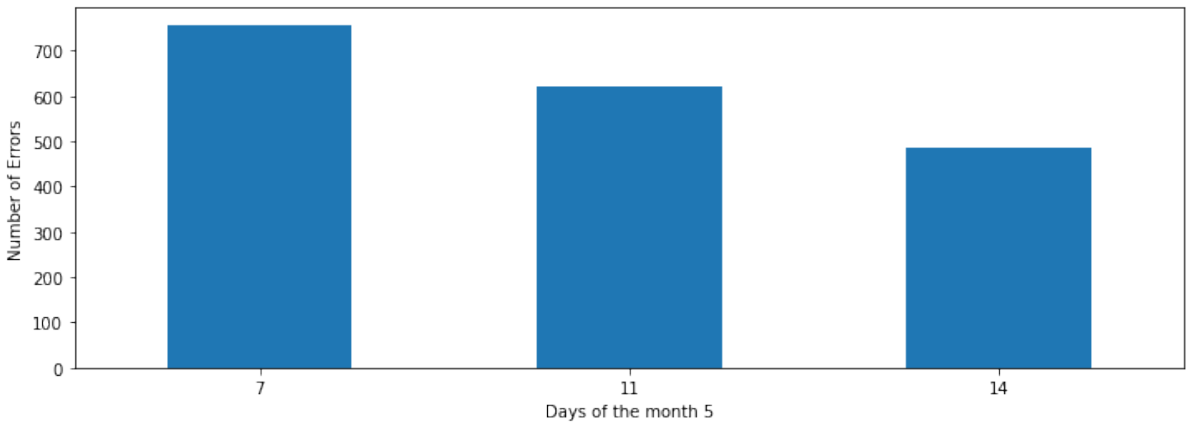

Figura 5.9: Numero di errori durante il mese di maggio nelle giornate di laboratorio del 7 (Classi e oggetti), 11 (Ereditarietà), 14 (Alberi binari).

5.3.2 Analisi degli errori nel tempo: giornate di laboratorio

Dopo aver analizzato la quantità di errori nei mesi e nelle settimane di laboratorio, sono state analizzate nello specifico le giornate di laboratorio ponendo particolare attenzione all'argomento trattato. Gli studi analizzati nei capitoli precedenti hanno evidenziato che vi sono specifici argomenti che risultano essere più complessi per gli studenti novizi. Questi argomenti generalmente sono la ricorsione e la programmazione ad oggetti. Inoltre, molti studi hanno anche dimostrato come gran parte degli studenti che ottengono risultati non soddisfacenti nei corsi di programmazione di base non hanno compreso a pieno argomenti di base come, ad esempio, il concetto di assegnamento. In questo caso i log sono stati raccolti solo a partire dal quarto laboratorio, dunque non è stato possibile analizzare gli errori dei primi laboratori che trattavano di argomenti basi-

lari come le variabili, l'assegnamento e le funzioni. I laboratori analizzati nello specifico, per quanto concerne gli errori, sono quelli in cui sono stati trattati argomenti di cui la letteratura ritiene essere complessi nell'apprendimento della programmazione. Il primo laboratorio analizzato nello specifico, con relativo argomento, sono stati:

- Lab 7: interazione indeterminata.
- Lab 9: ricorsione.
- Lab12: classi e oggetti.

5.3.3 Gli errori durante il laboratorio 7: iterazione indeterminata

Il laboratorio numero 7 corrisponde al laboratorio del 30 marzo. Durante questa giornata sono stati trattati argomenti riguardanti l'iterazione indeterminata. Si è scelto di analizzare questo laboratorio in quanto molti studi analizzati nel capitolo 2 hanno evidenziato come concetti come il ciclo while risulta essere spesso ostico per gli studenti, spesso viene confuso con il costrutto if o non viene utilizzato in maniera appropriata. La figura 5.10 mostra il numero di errori presenti nei log durante il laboratorio.

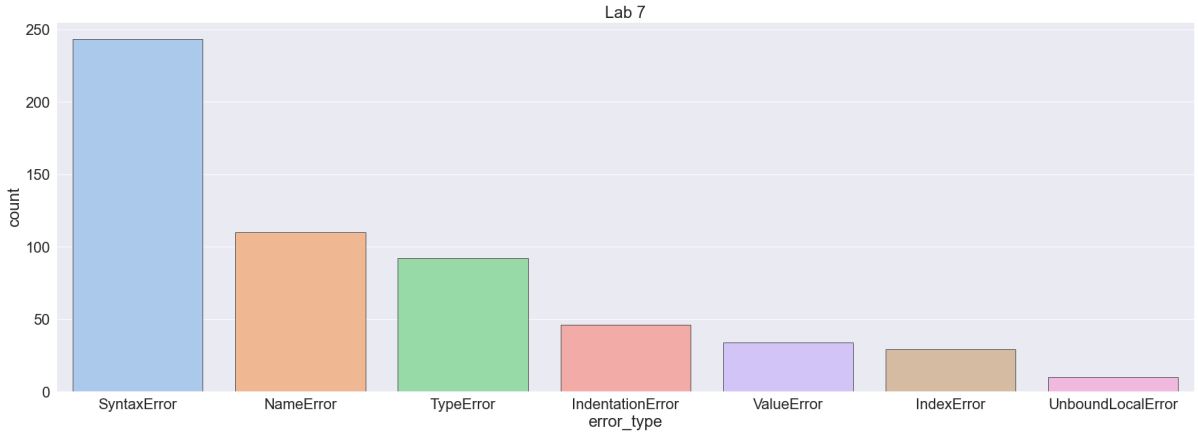

Figura 5.10: Numero di errori più frequenti durante il laboratorio 7 (Iterazione indeterminata).

In questo caso il numero maggiore di errori corrisponde al tipo SyntaxError, con circa 250 errori, una cifra molto alta. Come secondo errore più comune c'è il NameError e poco dopo TypeError. Un numero consistente da non sottovalutare è quello relativo agli errori di indentazione, ciò è molto probabilmente dovuto al fatto che gli studenti scrivevano in modo errato il ciclo while.

5.3.4 Gli errori durante il laboratorio 9: la ricorsione

Il laboratorio numero 9 corrisponde al laboratorio del 16 aprile. Durante questa giornata è stata trattata la ricorsione. Yarmish e Kopec [45] sostengono che l'individuazione e la classificazione dei tipi di errore che si trovano tipicamente nei programmi scritti dagli studenti è preziosa per gli istruttori in quanto consente ai docenti di concentrare le lezioni verso la minimizzazione dei malintesi tipici e il debug efficiente dei programmi degli studenti. Gli autori hanno notato che i problemi che richiedevano la ricorsione avevano la più alta percentuale di studenti che fraintendevano il problema. Yarmish e Kopec sostengono che quando si insegna la ricorsione, si dovrebbe prestare attenzione a spiegare chiaramente il ruolo dello stack ricorsivo. Molti studenti fraintendono la natura dello stack e, di conseguenza, scrivono programmi errati. Motivo per il quale, sono molto frequenti gli errori.
Per quanto riguarda l'analisi relativa al laboratorio 9, la figura 5.11, a differenza del laboratorio 7, mostra che il numero di errori di tipo TypeError sono molto più frequenti rispetto agli altri tipi. Il numero di questo tipo di errore supera i 250 eventi. Inoltre, è interessante notare che gli errori di tipo sintattico sono meno degli errori di tipo NameError. Il numero di errori di tipo TypeError sono il doppio del numero di errori di tipo NameError.

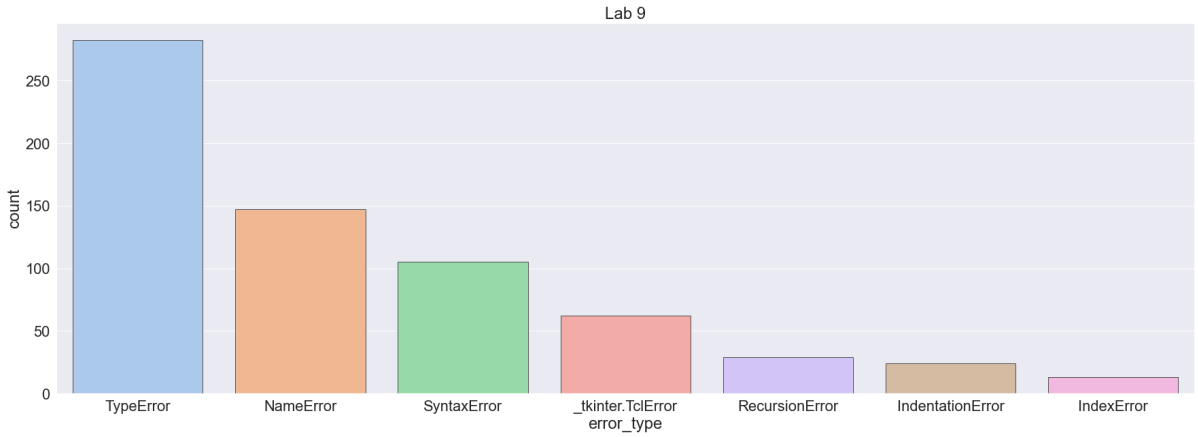

Figura 5.11: Numero di errori più frequenti durante il laboratorio 9 (Ricorsione).

5.3.5 Gli errori durante il laboratorio 12: classi e oggetti

L'ultimo laboratorio analizzato per l'analisi degli errori è stato il laboratorio 12. L'argomento trattato durante questo laboratorio è stata la programmazione ad oggetti. La programmazione ad oggetti è un altro argomento considerato dai principianti un compito molto difficile e per il quale trovano difficoltà nell'implementare programmi funzionanti. Gli oggetti, pur rappresentando il mondo reale, a un certo punto possono essere intangibili, vaghi e privi di contesto per i programmatori alle prime armi. Pillay [46] ha cercato di identificare gli errori commessi dagli studenti quando costruiscono per la prima volta programmi orientati agli oggetti in modo che questi possano essere presi in considerazione dai docenti durante la pianificazione di un primo corso sulla progettazione orientata agli oggetti. I risultati hanno mostrato che gli studenti hanno riscontrato le maggiori difficoltà nel comprendere alcuni dei concetti cardine della programmazione ad oggetti come l'ereditarietà e il polimorfismo (argomenti trattati principalmente nel laboratorio 13 e 12). Lo studio ha identificato quattro livelli di abilità necessari agli studenti per superare queste difficoltà di apprendimento, che possono essere riassunti in: conoscenza dei concetti, scomposizione dei problemi, identificazione delle relazioni tra le classi e comprensione dei concetti di livello superiore. Pillay ha, inoltre, individuato l'incapacità da parte degli studenti di scomporre un problema nelle sue parti costitutive.

Per questo motivo gli studenti dovrebbero essere esposti a diversi problemi per aiutarli a sviluppare l'abilità della decomposizione. In aggiunta a questo, agli studenti possono essere presentate scomposizioni di soluzioni errate che sono tenuti a correggere. Secondo Pillay è necessario fornire un feedback agli studenti mentre stanno risolvendo i problemi. Inizialmente, agli studenti possono essere assegnati problemi che sono già stati scomposti in modo da consentire agli studenti di concentrarsi sullo stabilire la relazione tra le classi e porre grande rilievo ad esercizi con feedback immediato, in quanto questa pratica aiuterebbero gli studenti a sviluppare una comprensione più profonda di questi concetti. L'analisi del laboratorio 12 può essere riassunta dalla figura 5.12. La figura mostra come in quanto caso il numero di errori più frequenti appartiene al tipo di errore NameError. Anche in questo caso, in maniera analoga al laboratorio 9, è interessante notare come il numero di errori relativi al tipo SyntaxError sia decisamente inferiore rispetto al numero di errori di tipo NameError. In questo caso, però, a differenza del laboratorio 9 è presente un tipo di errore in modo preponderante rispetto ai precedenti laboratori analizzati, ovvero l'errore di tipo AttributeError. L'attribute error è un tipo di eccezione che viene sollevata quando un riferimento ad un attributo o ad un assegnamento fallisce, quindi è un errore tipico della programmazione ad oggetti.

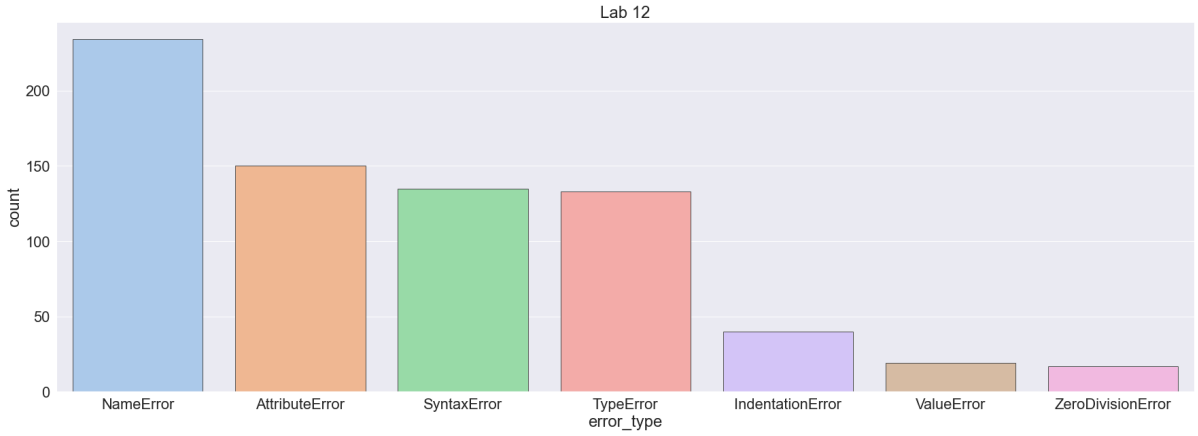

Figura 5.12: Numero di errori più frequenti durante il laboratorio 12 (Classi ed oggetti).

5.3.6 Lo studio del tipo di errore per migliorare la didattica della programmazione

L'analisi degli errori ha evidenziato che gli eventi corrispondenti agli studenti che hanno ottenuto voti superiore al 26 sono predominanti anche per quanto riguarda il numero degli errori. Ciò non sorprende dato che gli eventi appartenenti a questa categoria sono più del 60%. Inoltre, la maggiore quantità di errori evidenzia anche il fatto che una maggiore esercitazione da parte degli studenti porta a risultati migliori all'esame finale. L'errore ha dunque la sua importanza. Studiare nello specifico il tipo di errore, tuttavia, può essere fondamentale per comprendere il comportamento e il progresso degli studenti durante la fase di apprendimento della programmazione. Dall'analisi è emerso che gli errori più frequenti sono quelli di sintassi. Gli errori di tipo sintattici sono molto comuni e non rappresentano un grande problema per l'apprendimento della programmazione. In maniera analoga ai risultati mostrati nei paragrafi precedenti relativi allo studio [43], è inoltre emerso che i TypeErrors e i NameError non sono solo uno degli errori di runtime più frequenti, ma sono anche i quelli che si ripresentano più frequentemente negli esercizi successivi.

Il laboratorio in cui sono stati trattati argomenti come l'iterazione indeterminata ha avuto un numero molto consistente di errori di tipo sintattici, ma meno errori di tipo Name e Type. Inoltre, la quantità di errori sintattici non subisce grandi differenze quando si comparano sulla base della categoria target. Un dato interessante è stato quello relativo ai tipi di errori più frequenti dei laboratori 9 e 12. L'analisi degli errori relativa al laboratorio 9 ha mostrato un alto numero di errori di tipo TypeError. Essendo l'argomento del laboratorio 9 la ricorsione, i risultati mostrano che questo tipo di errore è molto comune tra gli studenti. Questa informazione può essere molto utile per docenti che si trovano ad insegnare la ricorsione a studenti novizi. Dall'analisi è infatti emerso che questo tipo di errore ha una frequenza molto alta solo per questo tipo di argomento. I docenti potrebbero dunque concentrare parte del tempo su attività di debug relative a questo tipo di errore. In modo analogo al laboratorio 9 anche i risultati del laboratorio 12 evidenziano una frequenza maggiore per un tipo di errore specifico per un determinato argomento. L'argomento del laboratorio 12 è stato la programmazione ad oggetti. Il tipo di errore più frequente è stato il NameError. Il NameError è un tipo di eccezione che viene

sollevata quando un nome locale o globale non viene trovato, dunque molto frequente durante la programmazione ad oggetti. In maniera analoga a quanto detto dallo studio [46] è necessario fornire un feedback agli studenti mentre stanno risolvendo i problemi al fine di migliorare la capacità da parte degli studenti di scomporre un problema nelle sue parti costitutive, elemento necessario per la programmazione ad oggetti.

5.4 Analisi dei Run

Dopo aver analizzato gli errori si è proceduto con l'analisi degli eventi che corrispondevano alle volte in cui gli utenti hanno eseguito il codice. Dall'analisi è emerso che:

- Il dataset contiene 19477 istanze relative all'evento run.
- La media del numero di run tra tutti gli studenti è di 209.43.

La figura 5.13 mostra il numero di run per ogni studente.

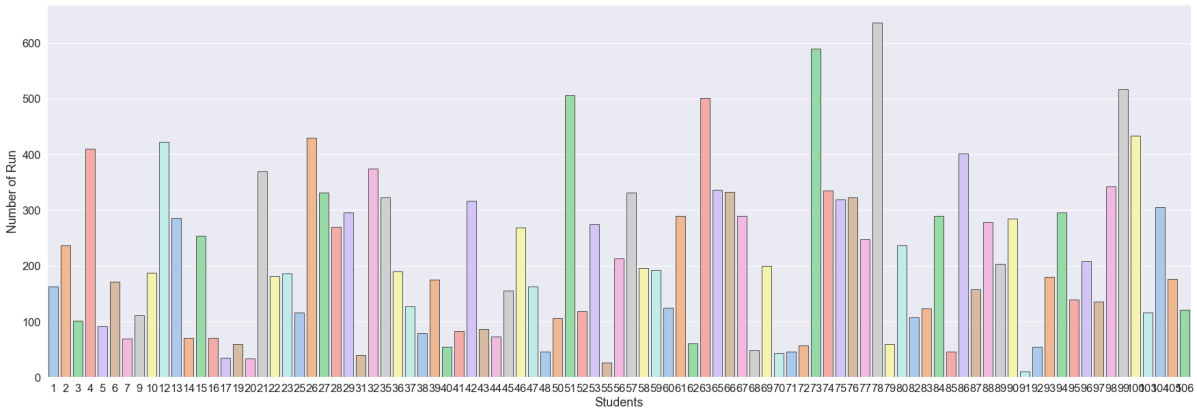

Figura 5.13: Numero totale di codici eseguiti per ogni studente.

5.4.1 Analisi dei Run nel tempo

Anche per l'analisi dei Run, come nell'analisi degli errori, i dati sono stati analizzati nel tempo. La figura 5.14 mostra la quantità dei run durante i mesi. Il mese di marzo e il

mese di aprile hanno registrato circa 8000 eventi relativi ai run. Il mese di maggio risulta essere il mese con meno eventi, poco meno di 4000. A differenza degli errori 5.5, il mese di marzo contiene molti eventi relativi alle volte in cui gli studenti hanno eseguito il codice. In proporzione ciò assume un importante significato, infatti, è interessante notare come nonostante il numero di errori sia molto più basso durante il mese di marzo rispetto ai successivi 2 mesi di laboratorio, ciò non è così per il numero di run. Gli studenti, quindi, durante il primo mese di laboratorio nonostante avessero eseguito il codice molte più volte rispetto ai mesi di aprile e maggio, hanno fatto un minor numero di errori.

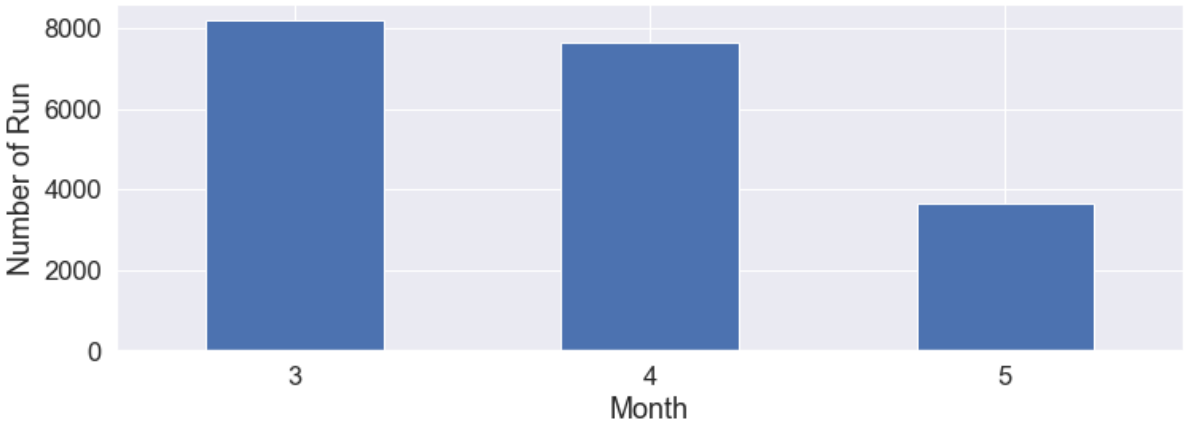

Figura 5.14: Numero di codici eseguiti nei mesi.

Numero dei run nelle settimane

La figura 5.15 mostra il numero di run nelle settimane, confermando il dato che vede il mese di marzo come il mese in cui sono stati eseguiti più programmi. Nello specifico è interessante notare come la settimana con il numero di run più alta sia la 13. La settimana 13 corrisponde ai laboratori 5 e 6, in cui sono stati trattati argomenti come l'iterazione determinata e il debugging. Uno dei motivi per il quale questa settimana ha registrato un alto numero di run è dovuto al fatto che durante il laboratorio gli studenti hanno utilizzato la libreria turtle.

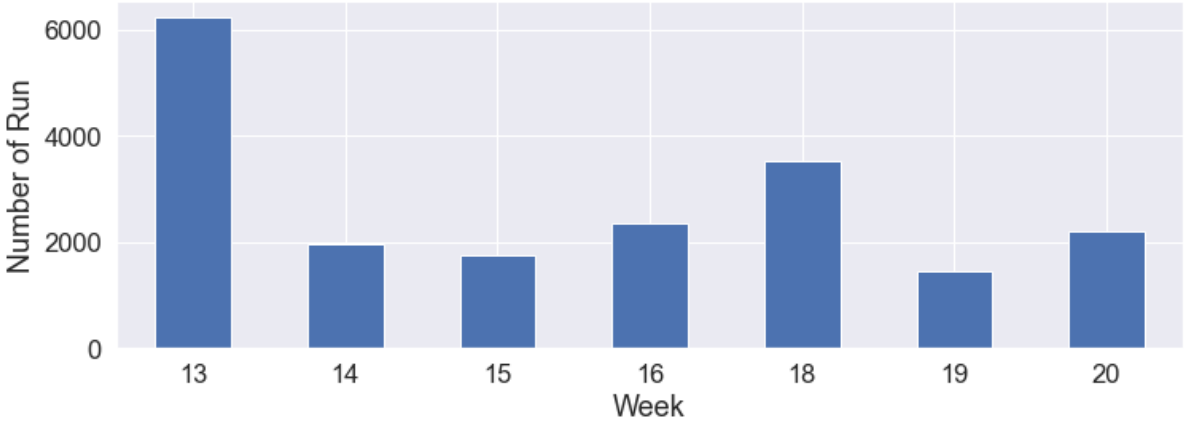

Figura 5.15: Numero di codici eseguiti nelle settimane.

Numero dei run nelle giornate di laboratorio

A dimostrazione di quanto detto, la figura 5.16 mostra che la giornata del 23 del mese di marzo registra poco più di 4000 eventi riguardanti i run. Come è possibile notare dalle figure relative ai mese di aprile e maggio 5.17 5.18, il numero di eventi relativi ai run è quasi sempre intorno ai 2000. Tuttavia, è interessante notare che il giorno 27 del mese di aprile il numero di run è molto più basso rispetto alle altre giornate di laboratorio dello stesso mese. La giornata del 27 corrisponde al laboratorio sui dizionari.

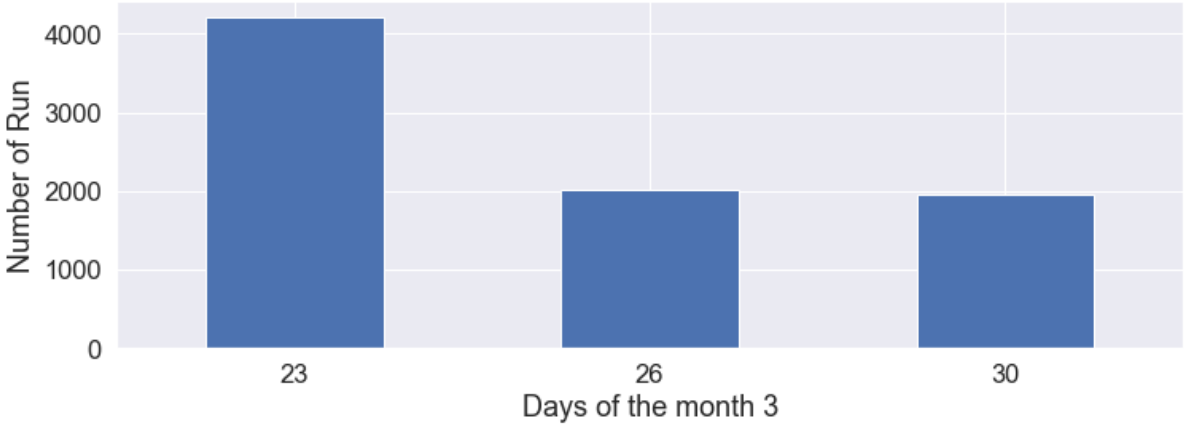

Figura 5.16: Numero di codici eseguiti durante il mese di marzo nelle giornate di laboratorio del 23 (Iterazione determinata), 26 (Debugging), 30 (Iterazione indeterminata).

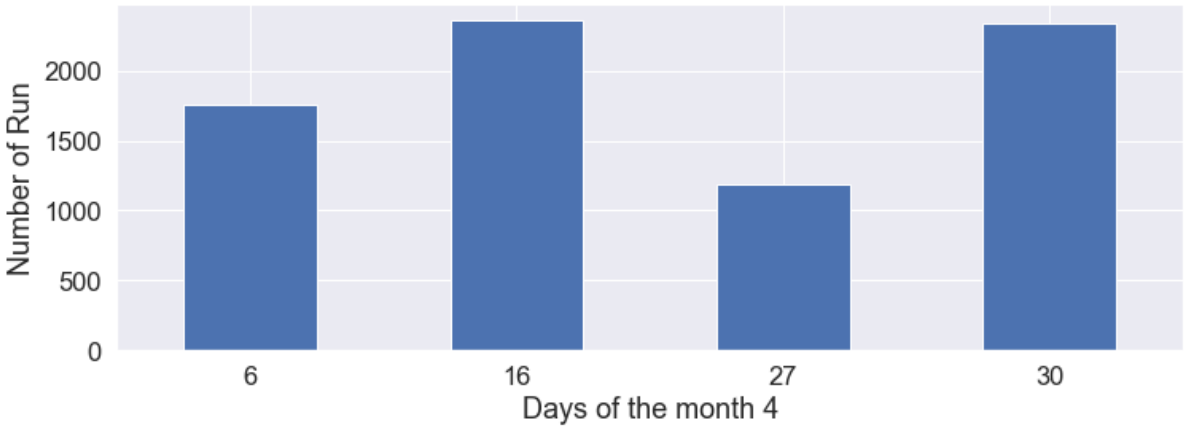

Figura 5.17: Numero di codici eseguiti durante il mese di aprile nelle giornate di laboratorio del 6 (Liste), 16 (Ricorsione), 27 (Dizionari), 30 (Comprehension).

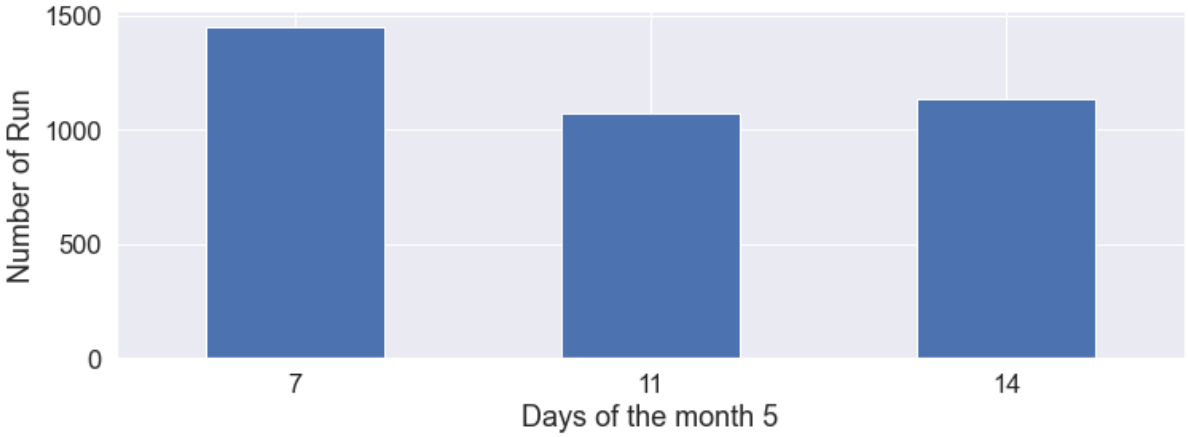

Figura 5.18: Numero di codici eseguiti durante il mese di maggio nelle giornate di laboratorio del 7 (Classi e oggetti), 11 (Ereditarietà), 14 (Alberi binari).

5.5 Analisi del codice incollato

Dopo aver analizzato gli eventi relativi ai run, si è proceduto con l'analisi degli eventi che corrispondevano alle volte in cui gli utenti hanno incollato del codice in Thonny. Dall'analisi è emerso che:

- Il dataset contiene 7494 istanze relative all'evento codice copiato.
- La media del numero di run tra tutti gli studenti è di 82.35.

La figura 5.19 mostra il numero di codice incollati per ogni studente, durante tutte le lezioni di laboratorio.

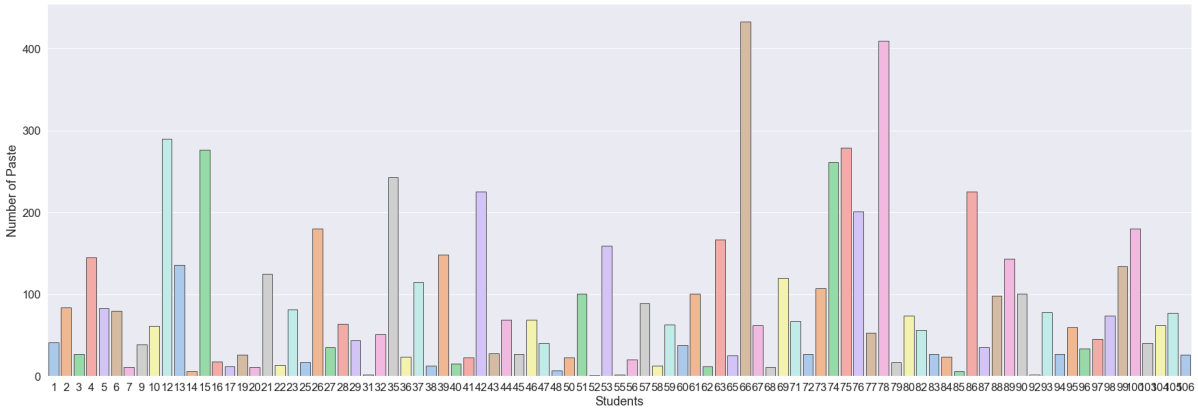

Figura 5.19: Numero di codici incollati per ogni studente.

5.5.1 Analisi del codice incollato nel tempo

L'analisi ha evidenziato che il mese in cui gli studenti hanno incollato più codice è stato aprile, come è possibile notare nella figura 5.20 nel mese di aprile sono stati registrati circa 3000 eventi relativi al codice incollato.

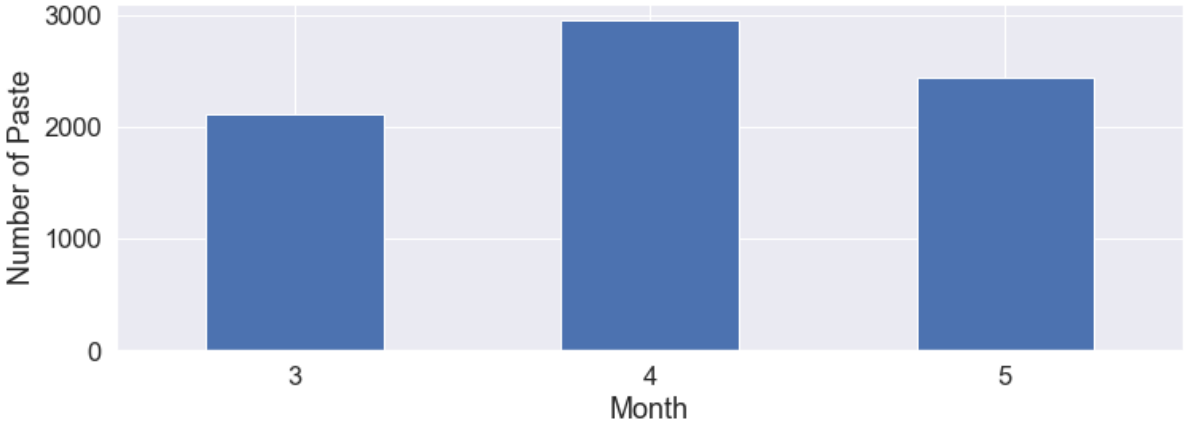

Figura 5.20: Numero di codici incollati nei mesi di laboratori.

I risultati dell'analisi relativi alle specifiche giornate di laboratorio sono mostrate nelle figure 5.21, 5.22 e 5.23.

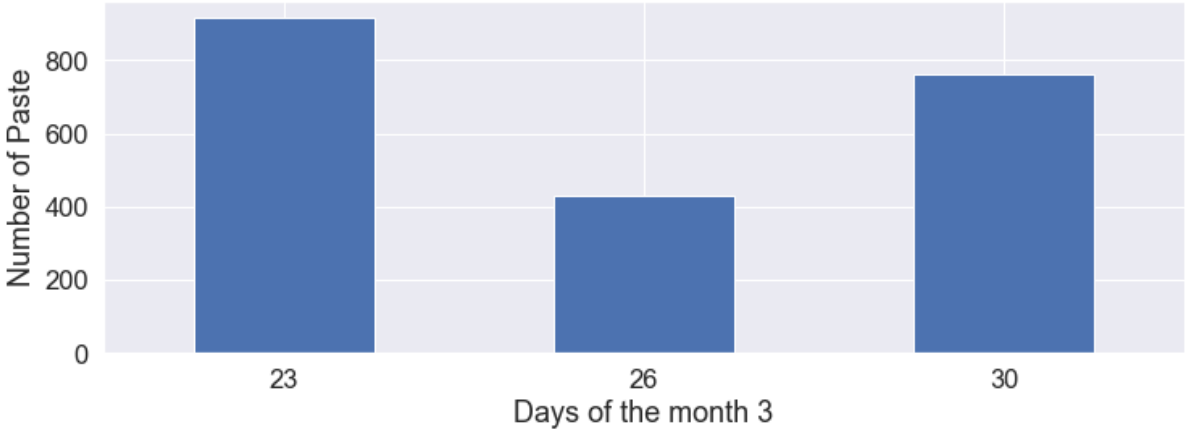

Figura 5.21: Numero di codici incollati durante il mese di marzo nelle giornate di laboratorio del 23 (Iterazione determinata), 26 (Debugging), 30 (Iterazione indeterminata).

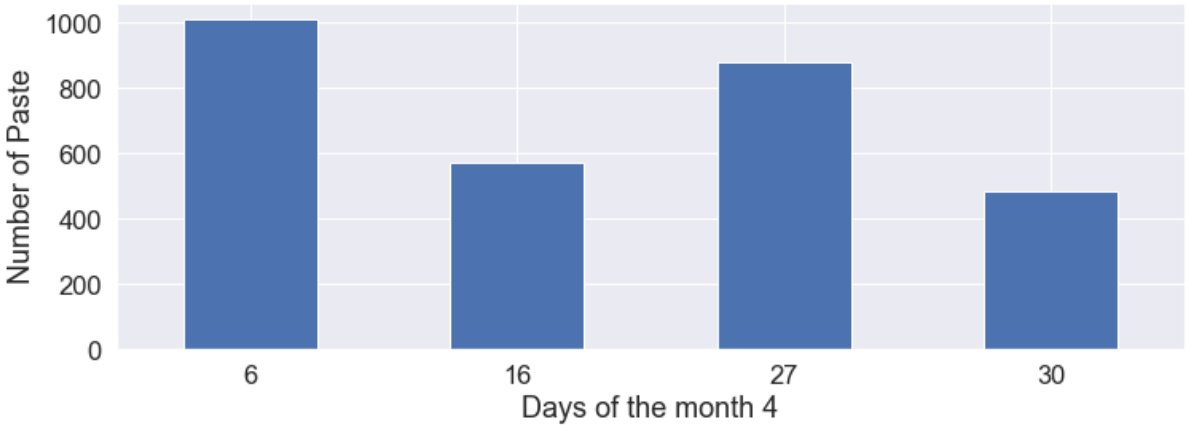

Figura 5.22: Numero di codici incollati durante il mese di aprile nelle giornate di laboratorio del 6 (Liste), 16 (Ricorsione), 27 (Dizionari), 30 (Comprehension).

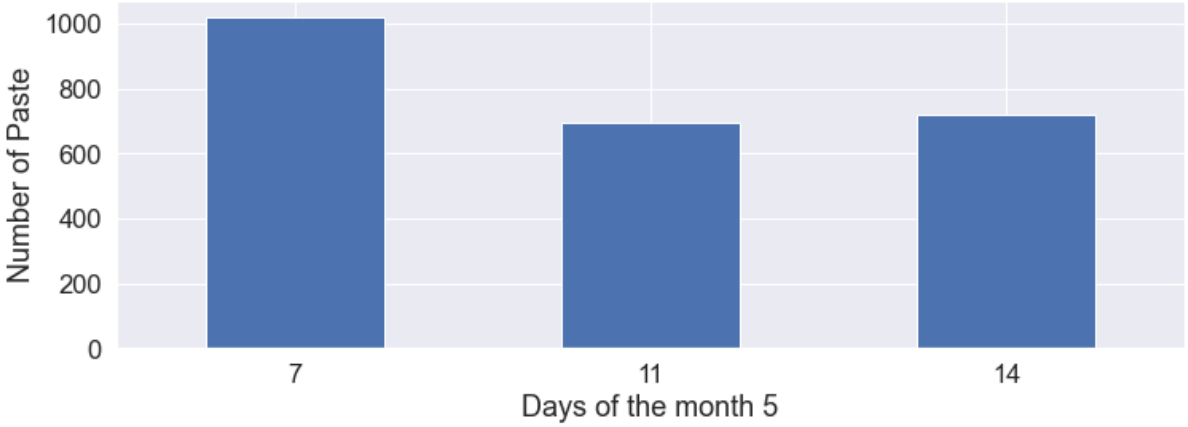

Figura 5.23: Numero di codici incollati durante il mese di maggio nelle giornate di laboratorio del 7 (Classi e oggetti), 11 (Ereditarietà), 14 (Alberi binari).

5.6 Studio delle correlazioni

Il passo successivo dell'analisi è stato quello di cercare delle correlazioni tra le features. Al fine di esprimere in maniera quantitativa l'intensità del legame tra due variabili è necessario infatti calcolare un indice di correlazione. Con i dati pre-processati è stato creato un nuovo dataset raggruppando i dati sulla base dell'ID di ciascuno studente. Questo dataset presenta sulle righe gli studenti. Le colonne sono rappresentate dagli errori più frequenti, il numero di run, il numero di volte in cui lo studente ha incollato del codice, il numero totale di eventi e lo score.
Il dataset è stato normalizzato colonna per colonna. Nella normalizzazione, i dati vengono ridimensionati su un intervallo fisso, in genere da 0 a 1. Questa è una buona tecnica da usare quando non si conosce la distribuzione dei dati o quando si sa che la distribuzione non è gaussiana (ossia non ha la forma di una curva a campana). Il metodo utilizzato per la normalizzazione è il Min-Max Scaling implementato con Scikit Learn.
Per lo score, invece, essendo una variabile di tipo categorica, è stato effettuato il *one-hot encode*. A differenza del *label encoding* che etichetta i valori di una variabile come un numero che parte da 1 per poi aumentare progressivamente, il one-hot encoding converte

i dati categoriali in una rappresentazione vettoriale binaria. In questo caso, dato che la variabile score può assumere 3 possibili valori, richiamando la funzione get_dummies specifica per il one-hot encode inpandas, sono state create tre colonne relative ai tre valori possibili per la features score.

5.6.1 Correlazione lineare di Pearson

Il primo indice utilizzato per il calcolo delle correlazioni è quello di Pearson. L'indice di correlazione di Pearson (anche detto coefficiente di correlazione lineare o coefficiente di correlazione di Pearson) tra due variabili statistiche è un indice che esprime un'eventuale relazione di linearità tra esse. L'indice di correlazione di Pearson è definito come la loro covarianza divisa per il prodotto delle deviazioni standard delle due variabili. Tale coefficiente può assumere valori che vanno da –1.00 (tra le due variabili vi è una correlazione perfetta negativa) e +1.00 (tra le due variabili vi è una correlazione perfetta positiva). Una correlazione uguale a 0 indica che tra le due variabili non vi è alcuna relazione.
La libreria pandas mette a disposizione la funzione corr. Specificando come indice quello di pearson, restituisce la matrice delle correlazioni tra le variabili, come è possibile vedere nella figura 5.24.

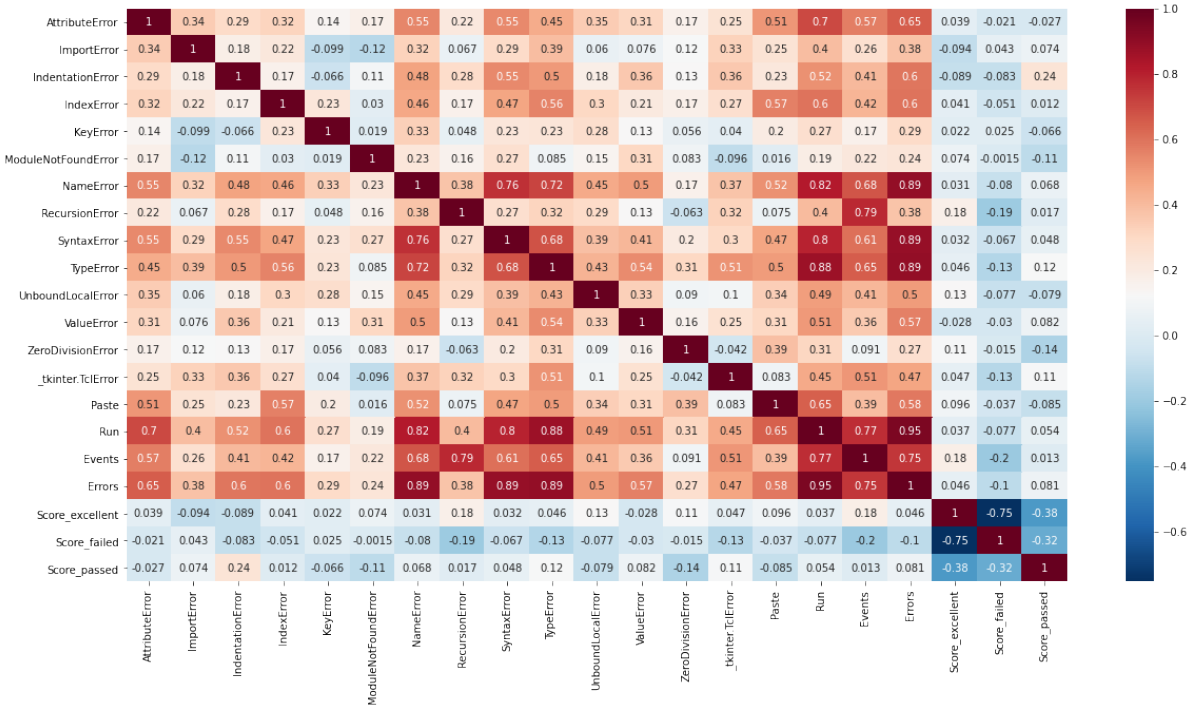

Figura 5.24: Heatmap delle correlazioni di Pearson.

5.6.2 Coefficiente di correlazione per ranghi di Spearman

Il secondo indice utilizzato per il calcolo delle correlazioni è quello di Spearman. L'indice di correlazione R per ranghi di Spearman è una misura statistica non parametrica di correlazione. Essa misura il grado di relazione tra due variabili e l'unica ipotesi richiesta è che siano ordinabili, e, se possibile, continue. Diversamente dal coefficiente di correlazione lineare di Pearson, il coefficiente di Spearman non misura una relazione lineare anche qualora vengano usate misure intervallari. Infatti esso permette di stabilire quanto bene una relazione tra due variabili può essere descritta usando una funzione monotona. Il segno della correlazione di Spearman indica la direzione dell'associazione tra X (la variabile indipendente) e Y (la variabile dipendente). Se Y tende ad aumentare quando X aumenta, il coefficiente di correlazione di Spearman è positivo. Se Y tende a diminuire quando X aumenta, il coefficiente di correlazione di Spearman è negativo. Una correlazione di Spearman uguale a zero indica che non vi è alcuna tendenza di Y ad aumentare

o diminuire quando X aumenta. L'indice di correlazione di Spearman cresce man mano che X e Y si avvicinano all'essere funzioni monotone perfette l'una dell'altra. Quando X e Y sono perfettamente monotonicamente correlati, il coefficiente di correlazione di Spearman è uguale a 1. Una perfetta relazione decrescente monotona implica che queste differenze hanno sempre segni opposti.
In modo analogo al calcolo delle correlazioni con Pearson, anche per l'indice di Spearman è stato possibile calcolare la matrice di correlazioni 5.25 tra le variabili del dataset mediante la funzione corr() messa a disposizione dalla libreria Pandas.

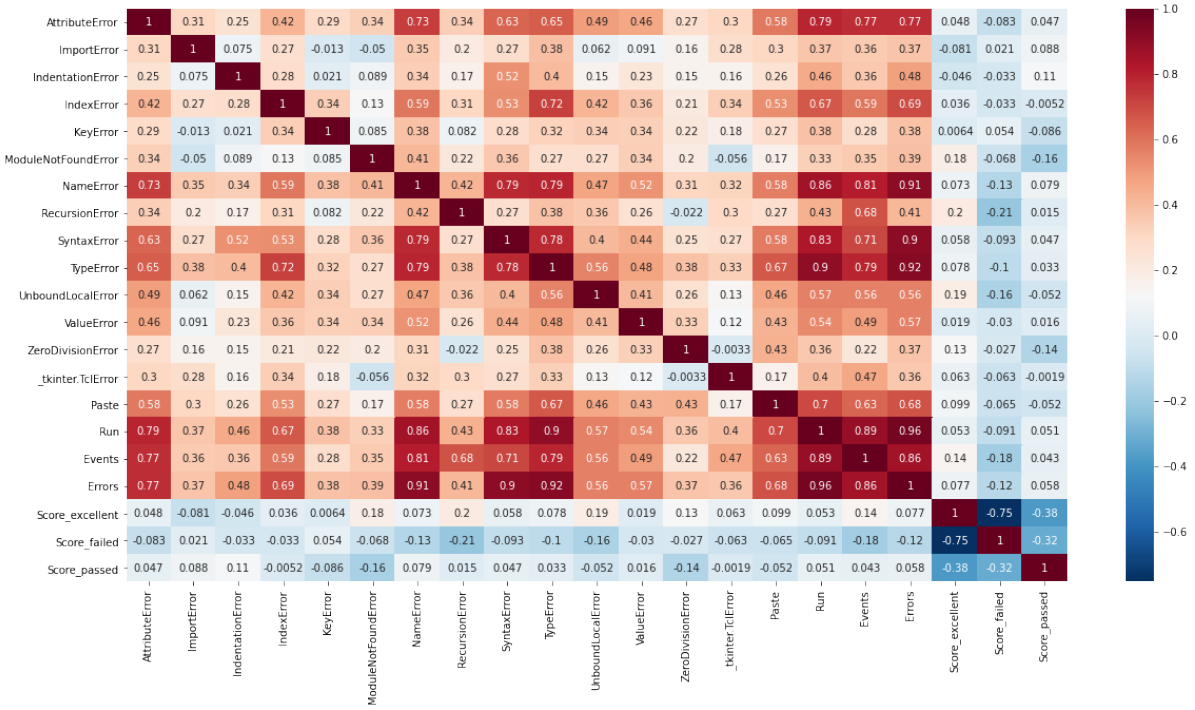

Figura 5.25: Heatmap delle correlazioni di Spearman.

5.6.3 Analisi delle correlazioni

Dall'analisi delle correlazioni sono emerse informazioni interessanti sulle correlazioni tra le features del dataset. Inoltre è stato anche calcolato il p-value per la matrice di correlazione come mostrato nella figura 5.26 . La figura 5.24 (relativa alla correlazio-

ne lineare di Pearson) mostra una correlazione di 0.18 tra la feature relativa allo score excellent e alla feature relativa al numero totale di eventi, con un p-value di 0.076. Tale risultato avvalora quanto detto nel paragrafo 5.1. Gli studenti che presentano un maggiore numero di eventi dei log sono gli studenti che hanno avuto i risultati migliori all'esame. Dunque è ragionevole pensare che coloro i quali si sono esercitati maggiormente, hanno mostrato una migliore capacità di apprendimento della programmazione dovuta al maggiore esercizio. Inoltre è interessante notare la correlazione tra l'errore RecursionError, tipico errore della giornata di laboratorio sulla ricorsione, con le diverse tipologie di score. Seppure non sia uno degli errori più frequenti, la correlazione tra lo score excellent e l'errore è di 0.2 con un p-value di 0.091 mentre la correlazione tra l'errore e le altre due tipologie di score ha un valore molto più basso. Essendo la ricorsione un argomento molto complesso per gli studenti novizi, quanto mostrato documenta che, in modo analogo alla correlazione tra il numero di eventi e i risultati eccellenti, gli studenti che si sono esercitati maggiormente e hanno fatto più errori sono quelli che hanno avuto un risultato migliore all'esame.

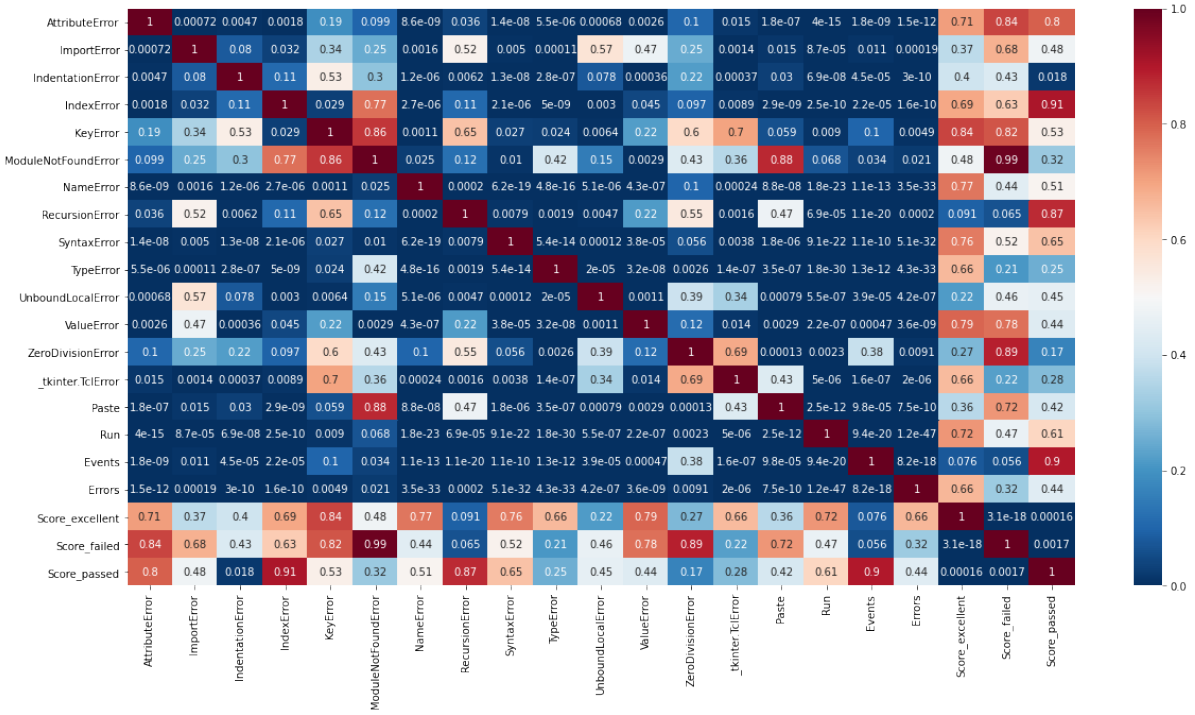

Figura 5.26: Heatmap dei p_value relativi alle correlazioni di Pearson.

Capitolo 6

Conclusioni

Gli studi analizzati dimostrano che imparare a programmare può essere complesso. La programmazione dunque pone sfide non solo allo studente ma anche al docente. Individuare come gli studenti comprendono le dinamiche della programmazione e come applicano i concetti teorici è un argomento di grande interesse per i docenti di informatica. L'interesse dei docenti che insegnano corsi introduttivi alla programmazione è quello di fornire un contesto di apprendimento che sia efficace ed utile all'apprendimento della programmazione per studenti che, molto probabilmente, non hanno mai avuto a che fare con linguaggi di programmazione. Alcuni studi dimostrano che molti studenti, anche dopo aver passato con successo l'esame di programmazione, mantengono solo una comprensione basilare e non approfondita della programmazione. Programmare, dunque, è complesso in quanto richiede un grande sforzo cognitivo. Nelle menti degli studenti novizi mancano i modelli mentali giusti. In questa tesi è stato discusso di quanto sia importante che i programmatori inesperti sviluppino modelli mentali corretti per imparare a programmare e per comprendere i concetti chiave teorici. Ogni corso introduttivo alla programmazione prevede una macchina concettuale. La macchina è implicita nel linguaggio di programmazione utilizzato. Il docente deve rendere l'apprendimento di una macchina concettuale un obiettivo esplicito attraverso una spiegazione elaborata allo scopo di illustrare la struttura di un sistema e il funzionamento interno. Lo scopo del docente deve essere, dunque, quello di evitare che lo studente crei un modello mentale scorretto. Lo studente che si è creato una macchina concettuale errata, infatti, potrebbe

essere convinto di saper spiegare come avviene il comportamento di alcuni programmi, anche se queste spiegazioni generalmente sono errati. Si è discusso, inoltre, delle misconcezioni e di quanto esse possano creare molti problemi durante la fase dell'apprendimento della programmazione e dell'importanza del tracciamento del codice, uno dei metodi per insegnare corretti modelli mentali e valide macchine concettuali. Sono diversi gli studi analizzati che dimostrano i benefici della didattica della programmazione mediante il tracciamento.

L'analisi svolta ha evidenziato che gli studenti che hanno avuto voti più alti negli esami, sono gli studenti che hanno generato molti più eventi nell'ambiente di sviluppo Thonny e dunque sono stati quelli che si sono esercitati di più e molto probabilmente hanno scritto più codice degli altri. L'analisi degli errori ha rilevato che gli errori più frequenti sono quelli di sintassi. Gli errori sintattici sono molto comuni, tuttavia - ci insegna la letteratura - non rappresentano un grande problema per l'apprendimento della programmazione. E' inoltre emerso che i TypeErrors e i NameError non sono solo uno degli errori di runtime più frequenti, ma sono anche quelli che si ripresentano più frequentemente negli esercizi dei laboratori successivi. Il laboratorio in cui sono stati trattati argomenti come l'iterazione indeterminata ha avuto un numero molto consistente di errori di tipo sintattici, ma meno per quanto riguarda errori di tipo Name e Type. Un dato interessante è stato quello relativo ai tipi di errori più frequenti dei laboratori sulla ricorsione e sulla programmazione ad oggetti. L'analisi degli errori relativa al laboratorio sulla ricorsione ha mostrato un alto numero di errori di tipo TypeError. Dall'analisi è infatti emerso che questo tipo di errore ha una frequenza molto alta solo per questo tipo di argomento. Tuttavia, la strutturazione dei log ha reso difficili e infruttuose analisi più specifiche per cercare di capire cosa abbia esattamente provocato queste differenze. In modo analogo al laboratorio sulla ricorsione, anche i risultati del laboratorio sulla programmazione ad oggetti evidenziano una frequenza maggiore per un tipo di errore specifico, ossia il NameError. L'elevata presenza di errori di tipo NameError durante questo laboratorio è, molto probabilmente, dovuto alla natura del paradigma stesso. Infine l'analisi delle correlazioni ha mostrato una correlazione tra la feature relativa allo score excellent e alla feature relativa al numero totale di eventi. Evidenziando, dunque,

che gli studenti che si sono esercitati maggiormente hanno una maggiore probabilità di comprendere meglio i concetti ed ottenere risultati migliori all'esame finale.

Durante l'analisi dei log generati dall'ambiente di sviluppo Thonny sono emerse alcune criticità relative alla struttura dei log. Ad esempio, Thonny non registra il testo scritto dagli studenti prima o dopo l'esecuzione del codice. A causa di ciò risulta essere complesso analizzare nello specifico i motivi degli errori oppure le dinamiche che hanno portato uno studente a non comprendere un concetto relativo alla programmazione. Tuttavia, potrebbe essere interessante per lavori futuri applicare degli algoritmi di machine learning ai log in modo da addestrare un sistema ad imparare come gli studenti agiscono durante le fasi di apprendimento.

Bibliografia

[1] S. A. Fincher and A. V. Robins, *The Cambridge handbook of computing education research*. Cambridge University Press, 2019.

[2] A. A. DiSessa, *Changing minds: Computers, learning, and literacy*. Mit Press, 2001.

[3] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[4] G. Weinberg, "The psychology of computer programming. 1971-1998."

[5] W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon, "Programming-languages as a conceptual framework for teaching mathematics," *ACM SIGCUE Outlook*, vol. 4, no. 2, pp. 13–17, 1970.

[6] D. C. Leonard, *Learning theories: A to z: A to Z*. ABC-CLIO, 2002.

[7] M. Lodi, *Introducing Computational Thinking in K-12 Education: Historical, Epistemological, Pedagogical, Cognitive, and Affective Aspects*. PhD thesis, Dipartimento di Informatica-Scienza e Ingegneria, Università di Bologna, 2020.

[8] R. C. Atkinson and R. M. Shiffrin, "Human memory: A proposed system and its control processes," in *Psychology of learning and motivation*, vol. 2, pp. 89–195, Elsevier, 1968.

[9] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming," *AcM SIGcSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007.

[10] C. Watson and F. W. Li, "Failure rates in introductory programming revisited," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pp. 39–44, 2014.

[11] A. Luxton-Reilly and A. Petersen, "The compound nature of novice programming assessments," in *Proceedings of the Nineteenth Australasian Computing Education Conference*, pp. 26–35, 2017.

[12] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," in *Working group reports from ITiCSE on Innovation and technology in computer science education*, pp. 125–180, 2001.

[13] S. Dehnadi, R. Bornat, *et al.*, "The camel has two humps (working title)," *Middlesex University, UK*, pp. 1–21, 2006.

[14] A. Robins, "Learning edge momentum: A new account of outcomes in cs1," *Computer Science Education*, vol. 20, no. 1, pp. 37–71, 2010.

[15] E. Patitsas, J. Berlin, M. Craig, and S. Easterbrook, "Evidence that computer science grades are not bimodal," pp. 113–121, 08 2016.

[16] L. E. Winslow, "Programming pedagogy—a psychological overview," *ACM Sigcse Bulletin*, vol. 28, no. 3, pp. 17–22, 1996.

[17] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of learning in novice programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 37–55, 1986.

[18] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *Acm sigcse bulletin*, vol. 37, no. 3, pp. 14–18, 2005.

[19] J. Meyer and R. Land, *Overcoming barriers to student understanding: Threshold concepts and troublesome knowledge*. Routledge, 2006.

[20] D. Shinners-Kennedy and S. A. Fincher, "Identifying threshold concepts: From dead end to a new direction," in *Proceedings of the ninth annual international ACM conference on International computing education research*, pp. 9–18, 2013.

[21] P. N. Johnson-Laird, *Mental models: Towards a cognitive science of language, inference, and consciousness*. No. 6, Harvard University Press, 1983.

[22] D. Gentner, "Stevens, al (eds.).(1983). mental models," 85.

[23] D. A. Norman, "Some observations on mental models," *Mental models*, vol. 7, no. 112, pp. 7–14, 1983.

[24] L. Ma, J. Ferguson, M. Roper, and M. Wood, "Investigating the viability of mental models held by novice programmers," in *Proceedings of the 38th SIGCSE technical symposium on computer science education*, pp. 499–503, 2007.

[25] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–64, 2013.

[26] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–64, 2013.

[27] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–24, 2017.

[28] K. Sanders and L. Thomas, "Checklists for grading object-oriented cs1 programs: concepts and misconceptions," *ACM SIGCSE Bulletin*, vol. 39, no. 3, pp. 166–170, 2007.

[29] R. E. Moreno and B. Park, "Cognitive load theory: Historical development and relation to other theories.," 2010.

[30] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive science*, vol. 12, no. 2, pp. 257–285, 1988.

[31] S. Gray, C. St. Clair, R. James, and J. Mead, "Suggestions for graduated exposure to programming concepts using fading worked examples," in *Proceedings of the third international workshop on Computing education research*, pp. 99–110, 2007.

[32] S. P. Davies, "Models and theories of programming strategy," *International journal of man-machine studies*, vol. 39, no. 2, pp. 237–267, 1993.

[33] V. Vainio and J. Sajaniemi, "Factors in novice programmers' poor tracing skills," *ACM SIGCSE Bulletin*, vol. 39, no. 3, pp. 236–240, 2007.

[34] M. Hertz and M. Jump, "Trace-based teaching in early programming courses," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, (New York, NY, USA), p. 561–566, Association for Computing Machinery, 2013.

[35] M. H. Goldwasser and D. Letscher, "Teaching an object-oriented cs1 -: With python," *SIGCSE Bull.*, vol. 40, p. 42–46, June 2008.

[36] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Trans. Comput. Educ.*, vol. 13, Nov. 2013.

[37] G. Alexandron, M. Armoni, M. Gordon, and D. Harel, "The effect of previous programming experience on the learning of scenario-based programming," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, (New York, NY, USA), p. 151–159, Association for Computing Machinery, 2012.

[38] C. Wilcox and A. Lionelle, "Quantifying the benefits of prior programming experience in an introductory computer science course," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, (New York, NY, USA), p. 80–85, Association for Computing Machinery, 2018.

[39] M. S. Kirkpatrick and C. Mayfield, "Evaluating an alternative cs1 for students with prior programming experience," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, (New York, NY, USA), p. 333–338, Association for Computing Machinery, 2017.

[40] A. Annamaa, "Thonny,: A python ide for learning programming," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, (New York, NY, USA), p. 343, Association for Computing Machinery, 2015.

[41] A. Annamaa, "Introducing thonny, a python ide for learning programming," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, (New York, NY, USA), p. 117–121, Association for Computing Machinery, 2015.

[42] D. McCall and M. Kölling, "A new look at novice programmer errors," *ACM Trans. Comput. Educ.*, vol. 19, July 2019.

[43] R. Smith and S. Rixner, "The error landscape: Characterizing the mistakes of novice programmers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 538–544, Association for Computing Machinery, 2019.

[44] P. Denny, J. Prather, and B. A. Becker, "Error message readability and novice debugging performance," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, (New York, NY, USA), p. 480–486, Association for Computing Machinery, 2020.

[45] G. Yarmish and D. Kopec, "Revisiting novice programmer errors," *SIGCSE Bull.*, vol. 39, p. 131–137, June 2007.

[46] N. Pillay, "A study of object-oriented design errors made by novice programmers," in *Proceedings of the 2009 Annual Conference of the Southern African Computer Lecturers' Association*, SACLA '09, (New York, NY, USA), p. 101–104, Association for Computing Machinery, 2009.