



UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

# Identificazione di smell linguistici in Infrastructure as Code

RELATORI

Prof. **Fabio Palomba**

Prof. **Dario Di Nucci**

Università degli Studi di Salerno

CANDIDATO

**Danilo Aliberti**

Matricola: 0512103836

Anno Accademico 2021-2022

*A Biagio e a nonna Caterina, le cui luci illuminano il mio percorso*

## Sommario

Gli antipattern linguistici sono cattive pratiche che riguardano inconsistenze linguistiche circa il *naming*, la documentazione e l'implementazione di un'entità software. Gli antipattern linguistici rendono il codice meno leggibile, poco manutenibile e poco chiaro. Ciò non è valido solo per il *codice classico*, cioè quello scritto per uno specifico linguaggio di programmazione, ma bensì si estende al *codice infrastrutturale*. L'impiego di codice infrastrutturale si è diffuso rapidamente, non solo nell'ambito business: esso è utilizzato nel processo di *Infrastructure as code*, il quale è impiegato nel *configuration management* di interi data centers. Con la diffusione del codice infrastrutturale, si sono propagate anche cattive pratiche di scrittura di tale codice. L'obiettivo della tesi è la valutazione dell'impiego del Machine Learning per la rilevazione delle inconsistenze linguistiche in Infrastructure as Code. Partendo da uno studio precedentemente condotto [4] che identifica un tipo di smell, si arriva a definire e analizzare un ulteriore smell linguistico caratteristico di IaC. Il focus del lavoro è uno specifico antipattern, secondo il quale vi è inconsistenza tra il nome di un task di *Ansible* e i parametri definiti all'interno di esso. A una serie di tasks collezionati da GitHub<sup>1</sup> è stata iniettata una mutazione dei loro parametri con una funzione sviluppata ad hoc per eseguire lo studio. Successivamente, sono stati allenati sei modelli di Machine Learning e Deep Learning per ognuna delle tecniche di word embedding scelte. Infine, dai risultati ottenuti dai classificatori e dalle tecniche di word embedding, si è giunti alla conclusione che il Machine Learning e il Deep learning possono essere utilizzati in maniera efficace per la rilevazione di inconsistenze linguistiche in IaC. Nonostante ciò, sono stati sollevati alcuni interrogativi che possono fungere da base per eventuali studi futuri sull'argomento proposto.

---

<sup>1</sup><https://github.com>

<b>Indice</b>	<b>ii</b>
<b>Elenco delle figure</b>	<b>iv</b>
<b>Elenco delle tabelle</b>	<b>v</b>
<b>1 Introduzione</b>	<b>1</b>
1.1 Motivazioni e Obiettivi . . . . .	1
1.2 Risultati . . . . .	2
1.3 Struttura della tesi . . . . .	2
<b>2 Background</b>	<b>3</b>
2.1 Infrastructure as Code . . . . .	3
2.1.1 Infrastructure as Code: definizione . . . . .	3
2.1.2 I problemi della gestione delle infrastrutture IT . . . . .	4
2.1.3 Il Cloud Computing: una possibile soluzione? . . . . .	5
2.1.4 Infrastructure as Code: il pezzo mancante del puzzle . . . . .	5
2.1.5 Come funziona Infrastructure as Code? . . . . .	6
2.2 Ansible . . . . .	8
<b>3 Stato dell'arte: Linguistic Antipattern e Code Smells in IaC</b>	<b>10</b>
3.1 Software Linguistic Antipatterns: definizione e classificazione . . . . .	11
3.1.1 Fa più di ciò che dichiara - <i>Does more than it says</i> . . . . .	11
3.1.2 Dichiara più di ciò che fa - <i>Says more than it does</i> . . . . .	12

---

3.1.3	Fa l'opposto - <i>Does the opposite</i> . . . . .	13
3.1.4	Contiene più di ciò che dichiara – <i>Contains more than it says</i> . . . . .	13
3.1.5	Dichiara più di ciò che contiene - <i>Says more than it contains</i> . . . . .	14
3.1.6	Contiene l'opposto – <i>Contains the opposite</i> . . . . .	14
3.2	Code smells in Infrastructure as Code . . . . .	14
3.2.1	Implementation Configuration Smells . . . . .	15
3.2.2	Design Configuration Smells . . . . .	16
3.2.3	Security Smells . . . . .	17
3.3	Opere correlate . . . . .	19
3.3.1	Qualità del codice IaC e Defect prediction . . . . .	19
3.3.2	Code smells e Linguistic Antipatterns nei tasks di Ansible . . . . .	19
3.3.3	Tool: FindICI, un detector di inconsistenze linguistiche . . . . .	20
<b>4</b>	<b>Studio empirico</b>	<b>24</b>
4.1	Definizione delle domande di ricerca . . . . .	24
4.2	Dataset . . . . .	25
4.3	Workflow alternativo e generazione della mutazione . . . . .	25
4.4	Selezione dei classificatori . . . . .	27
4.5	Selezione dei modelli . . . . .	27
4.6	Validazione dei modelli . . . . .	28
4.7	Selezione delle tecniche di Word Embedding . . . . .	29
4.8	Risultati . . . . .	30
4.9	Minacce alla validità . . . . .	34
<b>5</b>	<b>Conclusioni</b>	<b>36</b>
	<b>Ringraziamenti</b>	<b>38</b>

---

## Elenco delle figure

---

2.1	Esempio di un task Ansible . . . . .	9
3.1	Estratto di commit collegate a inconsistenze . . . . .	20
3.2	Panoramica sull'approccio di FindICI . . . . .	21
3.3	FindICI AST Model di un task . . . . .	22
4.1	Workflow alternativo di FindICI . . . . .	26
4.2	Task Ansible originale e task mutato . . . . .	26
4.3	Critical Difference diagrams basati su test di Wilcoxon-Holm per il detect di differenze significative tra le performance sui vari moduli Ansible. . . . .	32
4.4	Boxplots che mostrano MCC, AUC-ROC e Accuratezza per ogni classificatore. . . . .	33
4.5	Risultati ottenuti in termini di MCC, AUC ROC e Accuratezza con Word Embedding . . . . .	33
4.6	Critical Difference diagram basato sul test Wilcoxon-Holm per le differenze tra i valori MCC, AUC ROC, Accuratezza ottenuti applicando le tecniche di Word Embedding proposte . . . . .	34

---

## Elenco delle tabelle

---

3.1	Software Linguistic Antipatterns - Does more than it says . . . . .	12
3.2	Software Linguistic Antipatterns - Says more than it does . . . . .	13
3.3	Linguistic Antipatterns - Does the opposite . . . . .	13
3.4	Linguistic Antipatterns - Contains more than it says . . . . .	14
3.5	Linguistic Antipatterns - Contains the opposite . . . . .	14
3.6	Implementation configuration smells . . . . .	16
3.7	Design configuration smells . . . . .	17
3.8	Design configuration smells . . . . .	18
4.1	Distribuzione dei tasks Ansible nel dataset . . . . .	25
4.2	Risultati SVM per ogni modulo considerato . . . . .	30
4.3	Risultati MLP per ogni modulo considerato . . . . .	30
4.4	Risultati CNN per ogni modulo considerato . . . . .	31
4.5	Risultati RF per ogni modulo considerato . . . . .	31
4.6	Risultati XGBoost per ogni modulo considerato . . . . .	31
4.7	Risultati LSTM per ogni modulo considerato . . . . .	31

### 1.1 Motivazioni e Obiettivi

Questa tesi è il risultato di un lavoro sviluppato in un contesto internazionale, iniziato presso il **Jheronimus Academy of Data Science (JADS)** di 's-Hertogenbosch nei Paesi Bassi e conclusosi all'**Università degli Studi di Salerno** nel *SeSa Lab*. Ciò è stato possibile grazie al programma Erasmus+, il quale mi ha permesso di partecipare al programma di internazionalizzazione per condurre il tirocinio curricolare all'estero, presso il *JADE Lab* del JADS. Il JADS è un'organizzazione nata dalla cooperazione tra l'Università di Tilburg e quella di Eindhoven (TU/e) e offre master di specializzazione in Data Science e affini, oltre a collaborare in maniera attiva con i business per fornire supporto e spazio all'imprenditorialità innovativa basata sui dati. L'infrastructure as code (IaC) è diventata la soluzione definitiva nel development. IaC consente di soddisfare le continue ed esigenti modifiche dell'infrastruttura in maniera scalabile, efficiente e soprattutto tracciabile. Come suggerisce il nome, *Infrastructure as Code*, le configurazioni delle infrastrutture sono scritte sotto forma di codice vero e proprio, e sono *human readable*. Proprio per questo motivo, tale metodologia comporta molti vantaggi, ma porta con sé anche i problemi relativi alla forma del codice stesso, in particolare circa il *naming* e la documentazione. Essendo una tecnologia abbastanza giovane, pochi sono gli studi e le ricerche condotte in tale ambito e alcune di queste ricerche sono state svolte da parte di membri del *JADE Lab*, congiuntamente a componenti del *SeSa Lab*. È in questo background che si sviluppa il presente studio: partendo da una ricerca precedentemente effettuata [4], la



presente vuole essere un'estensione di essa ed ampliare l'analisi ad un antipattern linguistico più *atomico*: tale antipattern riguarda l'inconsistenza tra il *naming* di un task Ansible e i parametri definiti nel suo corpo.

## 1.2 Risultati

Come conferma di precedenti studi [4], dal presente studio emerge che il machine learning e il deep learning sono soluzioni più che valide per la produzione di classificatori il cui fine è la rilevazione di inconsistenze linguistiche, in particolar modo per l'antipattern analizzato. Nonostante ciò, il training dei modelli dipende da molte variabili e, quindi, potrebbe risultare in alcuni casi non preciso nell'individuazione di tali antipatterns. Al termine del lavoro e dall'analisi dei risultati, si è arrivati a porsi importanti domande circa l'impiego di tecniche di machine learning per la rilevazione di antipattern linguistici in IaC. Tali domande sono da considerarsi come base per studi futuri circa l'argomento trattato o come spunto per eventuali studi da parte di terzi.

## 1.3 Struttura della tesi

La tesi è strutturata in 5 capitoli, inclusi *Introduzione* (capitolo 1) e *Conclusioni*. Il Capitolo 2 è interamente dedicato al *background* e fornisce una panoramica sui problemi collegati alle infrastrutture IT e proponendo una parziale soluzione iniziale, fino ad arrivare a quella fornita da *Infrastructure as Code*. Il Capitolo 3 presenta lo stato dell'arte, introducendo i *Linguistic Antipatterns* nell'ambito dello sviluppo software, per poi arrivare ai code smells in IaC. Il Capitolo 4 formalizza gli obiettivi dello studio in *research questions*, descrive la metodologia utilizzata per rispondere a tali domande, mostrando i risultati. Inoltre, vengono individuate le minacce alla validità dello studio e le *mitigations* adottate per ridurre tali minacce. Infine, nel Capitolo 5, sono riportate le conclusioni e gli spunti per eventuali studi futuri.

Questo capitolo affronta in linea generale il concetto alla base di Infrastructure as Code, partendo dalla sua definizione, passando per l'identificazione dei problemi principali di una gestione manuale di infrastrutture IT. Si analizza una soluzione parziale per poi arrivare a quella fornita da Infrastructure as Code, introducendo brevemente i tools esistenti che permettono tale approccio. A completamento, si illustrano le principali funzionalità dello strumento al momento più usato in ambito IaC (i.e., Ansible).

## 2.1 Infrastructure as Code

### 2.1.1 Infrastructure as Code: definizione

Con Infrastructure as Code si intende un approccio alla gestione e al provisioning delle infrastrutture IT che è gestito interamente via codice, anziché in maniera manuale. I files di configurazione creati con IaC contengono le specifiche dell'infrastruttura e facilitano la modifica e la distribuzione delle configurazioni. IaC è una soluzione che garantisce la ripetibilità del provisioning dello stesso ambiente. IaC facilita la gestione della configurazione e permette di evitare modifiche non documentate. Il controllo di versione è un aspetto fondamentale dell'Infrastructure as Code e tutti i files di configurazione dovrebbero essere sottoposti a tale controllo, come qualsiasi altro file di codice sorgente software. Configurare un'infrastruttura con il codice permette la modularità dei componenti, i quali, attraverso l'automazione, possono essere combinati in modi differenti. Automatizzando i processi di

provisioning si esonerano gli sviluppatori dalla gestione di server, sistemi operativi, storage e altri componenti dell'infrastruttura, ogni volta che sviluppano o distribuiscono un'applicazione. Codificare l'infrastruttura significa creare un modello ripetibile e manutenibile, facilmente distribuibile sulle macchine interessate.<sup>1</sup>

### 2.1.2 I problemi della gestione delle infrastrutture IT

Storicamente, il management di una infrastruttura IT è stato sempre un processo manuale. I tecnici si occupavano fisicamente dell'installazione e della configurazione dei server. Una volta terminato il processo di configurazione delle macchine, come ad esempio il setup del sistema operativo e di tutte le applicazioni necessarie, si effettuava il deploy delle app. Naturalmente, questo processo manuale portava spesso a risultati inaspettati e ad una serie di problemi. Il primo grande problema è dato dal costo. Era necessario assumere molti professionisti esperti per eseguire determinati task, dall'ingegnere del network ai tecnici di manutenzione hardware. Tutte queste persone, ovviamente, dovevano essere pagate, ma anche gestite. Ne consegue che ciò portava ad una maggiore complessità strutturale e comunicativa all'interno dell'organizzazione. Ciò risultava, spesso, in una spesa economica consistente, senza considerare i costi di mantenimento dell'infrastruttura. Altri due problemi riguardano la scalabilità e la disponibilità. Alla fine, tutto si riduce a problemi di performance: poiché la configurazione manuale dei sistemi è molto lenta, le applicazioni spesso si trovano a dover affrontare picchi di accesso, mentre gli amministratori di sistema cercano disperatamente di configurare i server per la gestione del carico. Ciò, ovviamente, va ad impattare la disponibilità. Se l'organizzazione non disponesse di server di backup o addirittura data center, l'applicazione potrebbe non essere disponibile per lunghi periodi di tempo. Un altro problema riguarda il monitoraggio e la visibilità delle prestazioni. Una volta configurata e completata l'infrastruttura, è necessario monitorarla affinché ci si possa assicurare che funzioni in maniera ottimale. Quando si verifica un problema, come si individua esattamente da dove proviene? È la rete, il server o l'applicazione? Strumenti come Ntreao possono fornire piena visibilità sulle prestazioni dell'intera infrastruttura IT. Con il rilevamento la configurazione automatica di Ntreao è possibile assicurarsi che non ci siano blind spots nell'ambiente e la mappatura della piattaforma, la correlazione degli eventi e l'analisi automatizzata delle cause principali consentono di individuare esattamente il punto dove si è verificato un problema. Ultimo ma non meno importante è il problema dell'inconsistenza. Se

---

<sup>1</sup>Red Hat, *Introduzione all'automazione – Cosa si intende con Infrastructure as Code (IaC)?*, <https://www.redhat.com/it/topics/automation/what-is-infrastructure-as-code-iac>

ci sono diverse persone che implementano manualmente le configurazioni, le discrepanze saranno inevitabili.<sup>2</sup>

### 2.1.3 Il Cloud Computing: una possibile soluzione?

Il cloud computing è nato anche per alleviare alcuni dei problemi elencati in precedenza. Esso libera dalla costrizione di costruire e mantenere i propri data center, abbattendo i costi ad essi associati. Tuttavia, il cloud computing è tutt'altro che una panacea. Sebbene permetta di configurare rapidamente le infrastrutture, risolvendo gravi problemi come quelli correlati a disponibilità e scalabilità, non fornisce una risoluzione per quelli legati all'inconsistenza. Come detto in precedenza, quando ci sono più figure che eseguono le configurazioni, la probabilità di riscontrare inconsistenze a livello di configurazione sono elevate.<sup>2</sup>

### 2.1.4 Infrastructure as Code: il pezzo mancante del puzzle

Prendendo in considerazione la definizione di IaC data in precedenza, definiamo il suo punto chiave: prima della IaC, il personale IT avrebbe dovuto modificare manualmente le configurazioni per gestire la propria infrastruttura. Probabilmente, il modo più utilizzato per gestire ciò, era quello di sviluppare script usa e getta per automatizzare alcune attività. Con IaC, la configurazione dell'infrastruttura assume la forma in un semplice file di codice. Siccome è solo testo, è semplice da modificare, copiare e distribuire. Fino ad ora, sono stati trattati i problemi causati da un approccio manuale alla gestione dell'infrastruttura IT. È stato detto come il cloud computing sia una soluzione a parte di questi problemi; concludiamo sostenendo che IaC è l'ultimo pezzo del puzzle. Saranno di seguito analizzati alcuni dei vantaggi che le organizzazioni trarranno dall'adozione di una soluzione basata su IaC.<sup>2</sup>

**Velocità.** Il primo benefit significativo fornito da IaC è la velocità. Infrastructure as code consente di configurare rapidamente l'intera infrastruttura eseguendo uno script. È possibile farlo per ogni tipo di ambiente, dal development alla produzione, passando per lo staging, il QA e altro ancora. IaC può rendere efficiente l'intero ciclo di vita dello sviluppo del software.<sup>2</sup>

**Consistenza.** I processi manuali causano errori. Gli esseri umani non sono infallibili; la gestione manuale dell'infrastruttura IT risulterà in discrepanze, non importa quante attenzioni si pongono alla forma e agli standard di configurazione. IaC risolve in parte questo

---

<sup>2</sup>Carlos Schults, *What is Infrastructure as Code? How it works, best practices, tutorials*, <https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/>

problema, facendo in modo che i files di configurazione stessi siano l'unica "fonte di verità". In questo modo, è garantito che le stesse configurazioni vengano distribuite più e più volte, senza discrepanze.<sup>2</sup>

**Controllo di versione.** Poiché è possibile assegnare una versione ai files di configurazione generati con IaC, si ha una piena tracciabilità delle modifiche effettuate su ciascuna configurazione. Non sarà necessario "indovinare" le modifiche precedentemente effettuate e quando sono state effettuate.<sup>2</sup>

**Maggiore efficienza del ciclo di sviluppo software.** Utilizzando IaC è possibile distribuire l'architettura infrastrutturale in più fasi. Ciò rende lo sviluppo del software più efficiente, portando la produttività del team a nuovi livelli.<sup>2</sup> I developers potranno usare la IaC per creare e avviare ambienti sandbox, i quali permetteranno loro di sviluppare in completo isolamento e sicurezza. Lo stesso vale per i professionisti del QA, i quali possono avere copie perfette degli ambienti di produzione in cui eseguire i test. Infine, al momento della distribuzione, sarà possibile il push sia dell'infrastruttura sia del codice di produzione, in un singolo step.<sup>2</sup>

**Costi.** Uno dei principali vantaggi dell'IaC è senza dubbio la diminuzione dei costi di gestione dell'infrastruttura IT. Impiegando il cloud computing combinato con IaC, i costi si riducono drasticamente. Ciò perché non si deve tener conto delle spese riguardanti l'hardware, così come l'assunzione di personale preposto alla sua cura o l'affitto dello spazio per custodire un data center. Ogni qualvolta che un'organizzazione ha professionisti ben pagati che svolgono attività che è possibile automatizzare, esse stanno sprecando denaro. Tutta la loro attenzione dovrebbe essere rivolta alle attività che vanno ad aggiungere valore all'intera organizzazione. Ed è qui che le strategie di automazione, tra cui la IaC, tornano utili. Impiegando tali strategie, si liberano le figure professionali dall'esecuzione di tali attività manuali, lente e soggette a errori, in modo che possano concentrarsi su ciò che conta di più.<sup>2</sup>

### 2.1.5 Come funziona Infrastructure as Code?

Gli strumenti IaC possono variare per quanto riguarda la modalità di funzionamento, ma, generalmente, è possibile suddividerli in due gruppi principali:

- *Imperative Programming Tools*, il quale definisce una sequenza di comandi o istruzioni in modo che l'infrastruttura possa raggiungere il risultato finale.

- *Declarative Programming Tools*, secondo il quale si “dichiara” il risultato desiderato, ovvero invece di delineare esplicitamente una sequenza di passaggi di cui l’infrastruttura ha bisogno per raggiungere il risultato finale, l’approccio dichiarativo mostra come appare tale risultato.<sup>2</sup>

### Imperative Programming for IaC

Nella programmazione imperativa, si specifica un elenco di passaggi che lo strumento IaC deve seguire per eseguire il provisioning di una nuova risorsa. Si specifica allo strumento come creare ogni ambiente, utilizzando una sequenza imperativa di comandi. *Chef*<sup>3</sup> è un popolare tool IaC imperativo, ma anche *Ansible*<sup>4</sup> e *Salt*<sup>5</sup> hanno un supporto per la programmazione imperativa. La programmazione imperativa richiede, d’altra parte, maggiori conoscenze di scripting, perché è necessario scrivere comandi per ogni passaggio di provisioning. Ciò dà il controllo sul modo in cui si eseguono le attività dell’infrastruttura, il che è l’ideale quando si devono apportare piccole modifiche, ottimizzare per uno specifico scopo o tenere conto di stranezze del software. Gli script IaC imperativi sono spesso anche meno idempotenti: i passaggi predefiniti possono portare a risultati diversi a seconda dell’ambiente. Inoltre, gli script IaC imperativi sono così espliciti che un errore in un passaggio può causare il fallimento dell’intera operazione.<sup>6</sup>

### Declarative Programming for IaC

Nella programmazione dichiarativa, invece, si specificano il nome e le proprietà delle risorse dell’infrastruttura della quale si vuole eseguire il provisioning. Successivamente, lo strumento IaC calcola come ottenere da solo quel risultato finale. Ad esso si dichiara ciò che si vuole, ma non il modo in cui giungere al risultato finale. Alcuni esempi popolari di strumenti IaC che utilizzano il paradigma di programmazione dichiarativa, includono *Terraform*, *Puppet*, *Ansible*, *Salt* e *CloudFormation*. Tale approccio è molto popolare nella IaC. Si definisce lo stato finale desiderato della configurazione finale e la soluzione IaC capisce autonomamente in che modo arrivarci. La programmazione dichiarativa è altamente idempotante o ripetibile, il che

---

<sup>3</sup>Progress Chef – Automation Software for continuous Delivery of Secure Applications and Infrastructure - <https://www.chef.io/>

<sup>4</sup>Ansible - <https://www.ansible.com/>

<sup>5</sup>Salt - <https://repo.saltproject.io/>

<sup>6</sup>Copado, *Declarative vs imperative programming for infrastructure as code*, 2022, <https://www.copado.com/devops-hub/blog/declarative-vs-imperative-programming-for-infrastructure-as-code-iac>

significa che si possono eseguire i comandi IaC più e più volte e ottenere comunque lo stesso risultato. Il paradigma dichiarativo si adatta bene anche alla modifica delle configurazioni, perché i passaggi di provisioning dello strumento IaC non sono definiti in modo esplicito. Il più grande svantaggio dell'approccio dichiarativo è dato dal fatto che si rinuncia al controllo sui singoli passaggi del processo di provisioning. Inoltre, non è la scelta migliore per piccole correzioni e aggiornamenti che possono essere gestiti da un semplice script CLI. In questi casi, la programmazione dichiarativa può complicare le cose e rallentare il processo.<sup>6</sup>

## 2.2 Ansible

Ansible è un software utilizzato per l'automazione di macchine. Può gestire l'installazione e la configurazione di qualsiasi componente di sistema, così come la definizione di procedure di deploy automatizzate. *Ansible*, a differenza di altri tools quali *Chef* o *Puppet*, basa la sua filosofia chiave su una parola: semplicità. Creato nel 2012 da Michael DeHaan, autore di *Cobbler*, è balzato subito tra i sistemi di *configuration management* più utilizzati, grazie ad alcune caratteristiche fondamentali intorno a cui è stato sviluppato:

- non necessita altro che di un accesso SSH tra la macchina controllore e i nodi da controllare e tutto si gestisce con semplici file di testo (YAML);
- è scritto in Python, che lo rende multiplatforma e non dipendente dalla distribuzione, leggero e performante;
- non richiede la conoscenza di linguaggi di programmazione, in quanto la sintassi YAML usata per scrivere le istruzioni è di semplice lettura e comprensione;

Ha subito scalato le classifiche dei tools più utilizzati, tanto che è stata fondata la *Ansible Inc.* per commercializzare il supporto e sponsorizzare la soluzione e, successivamente, l'azienda è stata acquistata da *Red Hat*.<sup>7</sup>

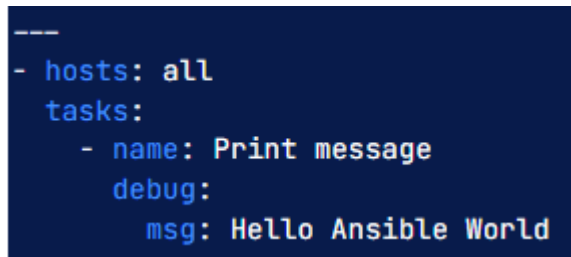
### Terminologia di Ansible

**Inventario.** Non è altro che una lista di macchine sulle quali Ansible può agire. Queste macchine possono essere raggruppate in modo da suddividerle per host. L'inventario può essere scritto sia in formato *ini* sia in YAML.<sup>7</sup>

<sup>7</sup>Michele Milanese, *Primi passi con Ansible*, 2018, <https://www.miamammauslinux.org/2018/02/primi-passi-con-ansible-parte-1/>

**Moduli.** I moduli sono comandi da eseguire sulle macchine. Ne esistono molteplici che vanno dall'installazione/rimozione di pacchetti alla gestione di servizi e molto altro ancora. La lista di moduli è davvero voluminosa ed è consultabile sulla documentazione ufficiale di Ansible. Ogni modulo ha il proprio set di proprietà e opzioni.

**Tasks.** I tasks sono operazioni che vengono eseguite sulle macchine. È possibile considerarli come una serie di comandi in sequenza che saranno eseguiti sulle macchine. Un task è il richiamo di un modulo con parametri particolari. I task di Ansible sono definiti all'interno di playbooks. Nella Figura 2.1 è riportato un task di esempio che stampa il messaggio tramite il modulo *debug*.<sup>7</sup>

The image shows a snippet of Ansible playbook syntax on a dark blue background with light blue and white text. It defines a task named 'Print message' that uses the 'debug' module to output 'Hello Ansible World'.---
- hosts: all
 tasks:
 - name: Print message
 debug:
 msg: Hello Ansible World

**Figura 2.1:** Esempio di un task Ansible

**Handlers.** Le operazioni collegate agli handler non vengono sempre eseguite ma richiamate come reazione ad un evento. Un esempio è quello di avviare un servizio dopo aver installato i packages necessari al suo funzionamento o riavviarlo dopo una modifica della sua configurazione.<sup>7</sup>

**Playbooks.** I playbook sono una lista di tasks, handlers e delle loro variabili. Tale lista è associata a gruppi di macchine di un inventario. I playbook sono utilizzati per avere un "manuale delle istruzioni" che Ansible andrà ad eseguire.<sup>7</sup>

**Ruoli.** I ruoli sono componenti dei playbook che hanno il compito di raggruppare operazioni che hanno uno scopo in comune. Queste vengono riunite per favorire la riusabilità. Ne è un esempio l'installazione, la configurazione e la gestione di un servizio riutilizzabile in diversi playbook.<sup>7</sup>



---

## Stato dell'arte: Linguistic Antipattern e Code Smells in IaC

---

Come affermato da Arnaoudova et al. [2], studi recenti e passati hanno mostrato che utilizzare un lessico poco chiaro all'interno di codice sorgente influisce negativamente sulla comprensibilità, manutenibilità e, in generale, sulla qualità del codice software prodotto. Oltre ad uno scarso utilizzo di termini adatti e documentazione, una descrizione poco chiara di un artefatto software può fuorviare i manutentori dalla sua reale funzione. Di conseguenza, gli sviluppatori impiegheranno più tempo e sforzi per comprendere porzioni di codice. Ciò potrebbe anche portare, come detto in precedenza, ad incomprensioni circa l'effettivo compito svolto da una parte di codice. Questo capitolo va a definire il concetto di Linguistic Antipattern applicato ai software e di Code Smells nell'ambito di Infrastructure as Code [28]. Mentre gli antipattern di design rappresentano apparenti soluzioni ricorrenti, gli antipattern linguistici non rappresentano altro che scelte di denominazione e descrizione imprecise o sbagliate. L'espressione code smells, invece, viene usata per indicare alcune caratteristiche che il codice può avere e che sono riconosciute come probabili indicazioni di un difetto di programmazione, andando ad abbassare la qualità del codice, a prescindere dal suo corretto funzionamento. Nell'ambito di IaC, andremo a trattare di Configuration Smells [29], in quanto il codice va a comporre i files di configurazione. Il capitolo seguente fornisce una categorizzazione degli antipattern linguistici applicati ai software e dei configuration smells, spiegando il tipo di incomprensione che possono causare. Infine, si applica il concetto di antipattern linguistico al contesto dell'Infrastructure Code.

### 3.1 Software Linguistic Antipatterns: definizione e classificazione

Arnaoudova et al. [2] forniscono una definizione chiara e precisa di Linguistic Antipattern: esso non è altro che una cattiva pratica riguardante le entità dei software, la loro nomenclatura e la loro documentazione. Una famiglia di antipattern linguistici su tutte è quella relativa alle incongruenze tra la denominazione, la documentazione e l'implementazione di un'entità software. In questo paragrafo sono presentate sei categorie di antipattern linguistici che riguardano specificamente il design del software: tre di esse riguardano il comportamento (metodi) e tre sono relative allo stato (attributi). Sono fornite le seguenti informazioni:

- Nome dell'antipattern linguistico;
- Descrizione.

Si sottolinea che la classificazione presente nei prossimi sottoparagrafi è un estratto del lavoro svolto da Arnaoudova et al. [2].

#### 3.1.1 Fa più di ciò che dichiara - *Does more than it says*

Come la denominazione suggerisce, ricadono in questa categoria gli antipattern linguistici che effettuano, ad esempio, più operazioni di ciò che indica il nome di un metodo o di una procedura. Di questa categoria fanno parte i seguenti antipattern:

Nome	Descrizione
<b>"Get" is more than an accessor</b>	Specificamente, in Java, i metodi accessori (chiamati anche getters), forniscono l'accesso in lettura agli attributi di una classe. Molto spesso, però, il getter effettua altre operazioni oltre a restituire l'attributo richiesto. Ogni altra azione dovrebbe essere documentata correttamente e, possibilmente, il naming del metodo dovrebbe modificato in qualcosa di diverso da, ad esempio, <code>getSomething</code> ;

<b>"Set" method returns</b>	In Java, i setters sono metodi che servono per impostare i nomi delle variabili. Di norma, essi non restituiscono alcun valore; quindi, questo antipattern si riferisce al fatto che, per qualche motivo, un setter restituisca un valore. Ciò deve essere necessariamente documentato e, ove possibile, si dovrebbe rinominare il metodo con un nome più consono alla sua funzionalità
<b>"is" returns more than a boolean</b>	quando un metodo ha il nome che inizia per "is", ci si aspetta che tale metodo restituisca un valore di tipo Boolean, con due possibili valori: true e false. Quindi, avere un metodo di questo tipo che non restituisce un valore booleano potrebbe risultare fuorviante. In questo caso, potrebbe essere necessario rinominare il metodo o almeno documentare il fatto che restituisce un valore differente da un boolean
<b>expecting but not getting a single instance</b>	quando il nome di un metodo indica che sarà restituito un singolo oggetto, dovrebbe essere coerente col suo tipo di ritorno. Se, invece, il tipo di ritorno è una collezione di oggetti e non un oggetto singolo, il metodo dovrà essere rinominato o ben documentato a riguardo

**Tabella 3.1:** Software Linguistic Antipatterns - Does more than it says

### 3.1.2 Dichiarare più di ciò che fa – *Says more than it does*

In questa categoria ricadono gli antipattern i cui nomi illudono circa la loro mansione:

Nome	Descrizione
<b>Not implemented condition</b>	si ha quando i commenti suggeriscono un comportamento condizionale, mentre il codice non lo implementa
<b>Validation method does not confirm</b>	si ottiene quando un metodo validatore non ritorna un valore per confermare la validazione

<b>"Get" method does not return</b>	mentre il nome del metodo getter suggerisce che si avrà qualche tipo di ritorno, il metodo stesso non restituisce alcun elemento
<b>Not answered question</b>	il nome del metodo è nella forma "is", che suggerisce un tipo di ritorno boolean, mentre il metodo non restituisce alcun valore booleano
<b>Transform method does not return</b>	il nome del metodo suggerisce che esso trasforma un oggetto, ma in realtà non restituisce alcun valore
<b>Expecting but not getting a collection</b>	il nome del metodo suggerisce che esso restituirà una collezione, ma esso ritorna o un singolo oggetto o nulla

**Tabella 3.2:** Software Linguistic Antipatterns - Says more than it does

### 3.1.3 Fa l'opposto - *Does the opposite*

Come il nome della categoria suggerisce, questa classe raggruppa gli antipattern che fanno l'opposto di ciò che un metodo dichiara.

Nome	Descrizione
<b>Method name and return type are opposite</b>	l'intenzione del metodo suggerita dal suo nome è in contraddizione con ciò che restituisce
<b>Method signature and comment are opposite</b>	la documentazione di un metodo è in contraddizione con la sua dichiarazione (nome e return type)

**Tabella 3.3:** Linguistic Antipatterns - Does the opposite

### 3.1.4 Contiene più di ciò che dichiara – *Contains more than it says*

Nome	Descrizione
<b>Says one but contains many</b>	si ha quando il nome di un attributo suggerisce che di esso sia presente una singola istanza, quando, in realtà, il suo tipo indica che l'attributo è una collezione di oggetti
<b>Name suggests boolean but type does not</b>	il nome di un attributo suggerisce che il suo valore sarà un boolean, ma la dichiarazione del suo tipo è diversa da Boolean

**Tabella 3.4:** Linguistic Antipatterns - Contains more than it says

### 3.1.5 Dichiarare più di ciò che contiene - *Says more than it contains*

- *Says many but contains one*: il nome di un attributo suggerisce che esso contenga una collezione di oggetti, ma il suo tipo indica che esso ne conterrà solo uno.

### 3.1.6 Contiene l'opposto – *Contains the opposite*

Nome	Descrizione
<b>Attribute name and type are opposite</b>	il nome di un attributo è in contraddizione col suo tipo
<b>Attribute signature and comment are opposite</b>	la documentazione dell'entità è in contraddizione con la sua dichiarazione

**Tabella 3.5:** Linguistic Antipatterns - Contains the opposite

## 3.2 Code smells in Infrastructure as Code

Il termine *Code Smell*, la cui definizione è fornita da Fowler nel suo libro [28], è usato per indicare una serie di proprietà che il codice sorgente può avere e che sono generalmente riconosciute come probabili segnali di un difetto di programmazione e progettazione. Si può facilmente intuire di come il concetto di Code Smell sia strettamente collegato a quello di *Linguistic Antipattern* precedentemente definito: entrambi non si riferiscono a bugs e non li rivelano, ma portano alla luce debolezze di progettazione che riducono la qualità del codice, a prescindere dall'effettiva correttezza del suo funzionamento. Nelle pratiche tradizionali di ingegneria del software, i bad smells sono classificati come *implementation (code) smells*, *design smells* e *architectural smells*, in base alla granularità dell'astrazione nel quale lo *smell* si presenta e colpisce [29]. Per quanto riguarda IaC, studi precedenti hanno descritto la qualità del codice infrastrutturale in termini di *smelliness* e tendenza ai difetti delle componenti di *Chef* e *Puppet*. Da un punto di vista di *smelliness*, Sharma T. et al [29], Spinellis et al. [28] e Rahman et al. [24] hanno catalogato i code smells di IaC in quattro categorie fondamentali [4]:

- *Implementation Configuration*, come, ad esempio, espressioni complesse e statements superati;

- *Design Configuration*, come gerarchie spezzate e blocchi duplicati;
- *Security smells*, come impostazione utente come admin di default e hard-coded secrets, ovvero codici di accesso scritti in chiaro all'interno di files di configurazione;
- *General smells*, di cui risorse eccessivamente lunghe e/o con numerosi attributi ne fanno parte.

### 3.2.1 Implementation Configuration Smells

Gli Implementation Configuration Smells sono problemi riguardanti la qualità del codice di configurazione, come ad esempio la naming convention, lo stile, la formattazione e l'indentazione [29]. Di seguito sono listati alcuni degli smells che ricadono in questo gruppo, con una breve descrizione.

Nome	Descrizione
<b>Inconsistent Naming Convention</b>	La scelta della naming convention devia da quella raccomandata dalla documentazione ufficiale del tool utilizzato
<b>Complex expression</b>	Un task contiene un'espressione o una serie di operazioni complesse e difficili da comprendere
<b>Duplicate Entry</b>	I parametri o le hash key duplicati/e presenti nel codice di configurazione
<b>Misplaced Attribute</b>	Il posizionamento errato degli attributi all'interno di una risorsa o di una classe (es. è consigliato specificare gli attributi obbligatori prima di quelli opzionali)
<b>Improper Alignment</b>	Il codice non è correttamente allineato o sono stati utilizzati caratteri di tabulazione
<b>Invalid property value</b>	Utilizzo di un valore non valido di una proprietà o di un attributo
<b>Incomplete tasks</b>	Il codice di configurazione presenta tasks con tags "FIXME" e/o "TODO", i quali indicano task incompleti
<b>Deprecated Statement Usage</b>	Il codice di configurazione usa uno degli statements deprecati
<b>Long Statement</b>	Il codice contiene statement eccessivamente lunghi

<b>Incomplete conditional</b>	Cioè quando il codice contiene un'istruzione condizionale non completa
-------------------------------	--

**Tabella 3.6:** Implementation configuration smells

### 3.2.2 Design Configuration Smells

I *Design Configuration Smells* rilevano problemi di qualità nel codice di progettazione di un modulo o di una struttura di un progetto di configurazione. È possibile reperire sulla documentazione ufficiale di qualsiasi tool IaC materiale da utilizzare per documentarsi sulle best practices per il design e la progettazione di files di configurazione, proprio per evitare problemi di natura smelly. Questo sottoparagrafo contiene una lista di design configuration smells, definita da Sharma et al. [29], ognuno accompagnato da una breve descrizione.

Nome	Descrizione
<b>Multifaceted Abstraction</b>	Ogni astrazione (i.e. una risorsa, una classe o un modulo) dovrebbe essere progettata per specificare le proprietà di un singolo pezzo di software. In altre parole, ogni astrazione dovrebbe seguire il principio di singola responsabilità. Un'astrazione soffre di un'astrazione multiforme quando gli elementi dell'astrazione non sono coesi.
<b>Unnecessary Abstraction</b>	Una classe o un modulo deve contenere dichiarazioni o statements che specificano le proprietà del sistema desiderato. Una classe vuota o un modulo mostra la presenza di tale smell e dev'essere necessariamente rimossa/o.
<b>Missing Abstraction</b>	Le dichiarazioni delle risorse e degli statements sono facili da fare e riutilizzare quando sono incapsulate in un'astrazione, come la classe o il 'define'. Un modulo soffre di tale smell quando le risorse e gli elementi del linguaggio sono dichiarati e usati senza essere incapsulati in un'astrazione.
<b>Insufficient Modularization</b>	Un'astrazione soffre di tale smell quando essa è larga e complessa e, di conseguenza, potrebbe essere modularizzata.

<b>Duplicate Block</b>	Un blocco duplicato di statements indica che, probabilmente, non è stata applicata un'astrazione adatta.
------------------------	--

**Tabella 3.7:** Design configuration smells

### 3.2.3 Security Smells

I *Security Smells* sono pattern di codice ricorrente che rappresentano indicatori di debolezze relative alla sicurezza che richiedono una particolare attenzione. Gli smells catalogati da Rahman et al. [24] riguardano i tools *Ansible* e *Chef* e sono elencate nella Tabella 3.8.

Nome	Descrizione
<b>Admin by default</b>	Questo smell è un modello ricorrente per specificare gli utenti predefiniti come admin. Tale smell può violare il principio del privilegio minimo, che raccomanda di progettare e implementare un sistema in modo tale che, per impostazione predefinita, venga fornito a qualsiasi entità la quantità minima di accesso necessaria
<b>Empty password</b>	Questo smell è un modello ricorrente di utilizzo di una stringa di lunghezza zero per una password. Una password vuota è indice di una password indubbiamente debole
<b>Hard-coded secret</b>	Questo smell riguarda la rivelazione di informazioni sensibili, come l'username e la password di qualche servizio, all'interno di scripts IaC. Tali scripts offrono l'opportunità di specificare intere configurazioni di sistema, come la configurazione delle credenziali di accesso, l'impostazione delle chiavi SSH per gli utenti e la specifica dei files di autenticazione



<b>Missing Default in Case Statement</b>	Tale smell è il pattern ricorrente riguardo la non gestione di tutte le combinazioni di input quando si implementa una logica condizionale. A causa di questo pattern, un utente malintenzionato può indovinare un valore (che non è gestito dalle istruzioni condizionali del caso) e causare un errore. Questo errore potrebbe fornire all'attaccante informazioni per il sistema in termini di stack traces o un system error
<b>No integrity check</b>	Questo pattern riguarda il download di contenuti da internet e il non controllo del materiale scaricato utilizzando il checksum o le firme gpg. Non controllando l'integrità, il developer presume che il contenuto sia sicuro e non sia stato compromesso da un potenziale aggressore. Il controllo d'integrità fornisce un ulteriore livello di sicurezza per garantire che il contenuto scaricato sia intatto
<b>Suspicious comment</b>	Questo smell riguarda l'inserimento di informazioni circa la presenza di difetti, funzionalità mancanti o debolezze del sistema all'interno dei commenti del codice di configurazione. Parole chiave come "TODO" e "FIXME" nei commenti vengono utilizzate per specificare un caso limite o un problema
<b>Use of weak cryptography algorithms</b>	Tale smell riguarda l'utilizzo di un algoritmo di crittografia debole, vale a dire MD5 e SHA-1. È risaputo che MD5 soffre di problemi di sicurezza, come dimostrato dal malware Flame nel 2012. Anche SHA-1 è suscettibile ad attacchi di collisione, quindi, l'utilizzo di algoritmi deboli per l'hashing potrebbe portare ad una violazione e a problemi di sicurezza

**Tabella 3.8:** Design configuration smells

### 3.3 Opere correlate

#### 3.3.1 Qualità del codice IaC e Defect prediction

La maggior parte dei lavori svolti in precedenza descrive la qualità del codice infrastrutturale in termini di *smelliness*, come, ad esempio, gli studi condotti da Sharma et al. [29]. Per quanto riguarda la parte di *defect prediction*, Rahman et al. [26] hanno identificato dieci misure di valutazione del codice che sono strettamente correlate con un codice IaC difettoso [4]. Dalla Palma et al. [9; 10; 11] hanno proposto un insieme di tools per calcolare le metriche di qualità per gli script Ansible e i progetti e, successivamente, per utilizzarle al fine del predict di script difettosi. Kumara et al. hanno proposto, invece, un tool per il detect di code smells negli script di TOSCA; Cito et al. [27] sono riusciti ad effettuare il detect delle violazioni delle best practices di Docker. Infine, Borovits et al. [4] hanno contribuito alla ricerca proponendo un approccio innovativo e automatizzato che utilizza le tecniche del word embedding e del learning per scovare gli antipattern linguistici in IaC, concentrandosi principalmente su inconsistenze di tipo *task name-task body* all'interno di scripts di Ansible. Tale scelta è stata effettuata per il fatto che Ansible, in industria, risulta essere il più utilizzato [14].

#### 3.3.2 Code smells e Linguistic Antipatterns nei tasks di Ansible

Dopo una panoramica sui *Code Smells* di IaC, passiamo alla definizione di Code smells e Linguistic Antipattern nel dominio di Ansible. Come detto in precedenza, Ansible è attualmente uno dei tools più utilizzati nell'ambito del configuration management [14], ma non è esente da inconsistenze, anzi: ciò lo espone ancora di più. Illustrando le commit relative alla Figura 3.1, Borovits et al. [4] hanno messo in luce che, benché i developers si sforzino per seguire le best practices, qualche snippet può sempre sfuggire al controllo della forma. In questi due tasks, ad esempio, i nomi sono in contraddizione con ciò che realmente fanno. Ad esempio, se il valore del parametro *state* del modulo *homebrew* è *absent*, questo sarà disinstallato. Sempre nella Figura 3.1, è presente un task con un'inconsistenza *nome-corpo* rilevante: il primo task (a destra) installa *nginx*, mentre il nome indica che sarà installato il package *supervisor*. In questo caso, l'inconsistenza denota un task *buggato* [4]. Tali inconsistenze di tipo *nome-corpo* dei tasks, possono essere considerate antipattern linguistici [2]. Come detto in precedenza, tali antipattern possono confondere gli sviluppatori/manutentori, siccome essi sono portati a supposizioni errate circa il comportamento del codice oppure a dedicare tempo e inutili sforzi per capirlo e chiarirlo [1]. Di conseguenza, risulta essenziale evidenziare la loro presenza, affinché possa essere prodotto codice chiaro e di qualità. In Ansible, il nome e

The image shows a screenshot of a GitHub commit history. It lists three commits with their titles and authors. Below each commit, there is a list of task names and their bodies, with some lines highlighted in green to indicate inconsistencies.

Commit 1: `linux-system-roles/logging`  
 Title: `Fixing the inaccurate task names in the sub roles.`  
 Author: `nhosoi` committed on Apr 27, 2020 ✓

Commit 2: `nusenu/ansible-relayor`  
 Title: `minor: fix unclear task name`  
 Author: `nusenu` committed on Aug 9, 2017

Commit 3: `OpenConext/OpenConext-deploy`  
 Title: `Fixed a typo in the name of a task`  
 Author: `jweewer` authored and `quartje` committed on Dec 28, 2018

Task names and bodies (from Commit 3):

- `- name: "ensure composer is installed"`
- `+ name: "ensure composer is not installed via homebrew"`
- `homebrew:`
- `name: composer`
- `state: absent`
- `- name: Ensure Kibana is started`
- `+ name: Ensure Kibana is started and enabled`
- `service:`
- `name: kibana`
- `enabled: true`

Task names and bodies (from Commit 2):

- `- name: install supervisor`
- `- apt: pkg=nginx state=present`
- `+ apt: pkg=supervisor state=present`
- `notify: enable supervisor`
- `- name: template Generate /etc/powerman/powerman.conf`
- `template:`
- `src: powerman.conf.j2`
- `- dest: powerman/powerman.conf`
- `+ dest: /etc/powerman/powerman.conf`
- `- name: Linux | Agent registration via rest-API`
- `block:`
- `- name: Retrieving authd Credentials`
- `- include_vars: authd_pass.yml`
- `+ name: Retrieving rest-API Credentials`
- `+ include_vars: api_pass.yml`

**Figura 3.1:** Estratto di commit collegate a inconsistenze

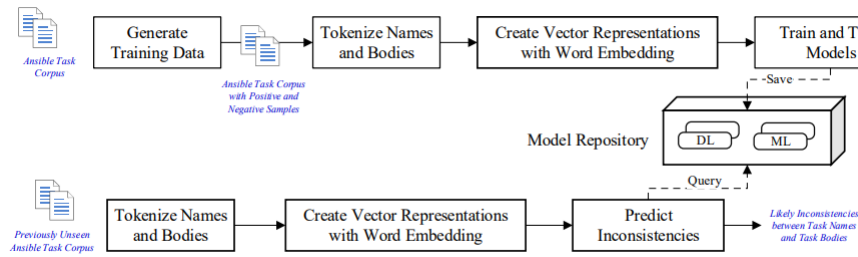
il corpo di un task differiscono totalmente da quelli dei metodi regolari dei classici linguaggi di programmazione. Il nome del task, in Ansible, è rappresentato da una frase completa o un frammento di frase, mentre il corpo è una configurazione specifica di un modulo Ansible. Per il detect di inconsistenze linguistiche di Ansible è nato *FindICI*, un tool sviluppato da Borovits et al. [4] nel contesto del detect di inconsistenze linguistiche nei tasks di Ansible.

### 3.3.3 Tool: FindICI, un detector di inconsistenze linguistiche

FindICI è un approccio innovativo presentato da Borovits et al. [4] per identificare le inconsistenze tra le descrizioni dei tasks in IaC (Ansible in particolare) in linguaggio naturale e i loro corpi [4]. In Figura 3.2 è sintetizzato il workflow dello strumento. La ricerca di un numero sufficiente di *buggy tasks* reali è un'attività impegnativa e dispendiosa, quindi FindICI applica delle trasformazioni del codice per generare un corpo di tasks inconsistenti [4]. Sia i nomi che i corpi sono tokenizzati e convertiti nelle loro rappresentazioni vettoriali, utilizzabili da un algoritmo di apprendimento. Poi, il tool addestra e valuta i classificatori binari utilizzando diversi algoritmi di machine learning e deep learning. I classificatori potranno rilevare inconsistenze di tipo *nome-corpo* dei tasks di Ansible ancora *unseen* basati su moduli [4]. Tali moduli saranno tokenizzati come il dataset utilizzato per il training [4].

### Generazione dati Training e Test

Il problema considerato è una classificazione binaria supervisionata. Quindi, il dataset necessario deve includere esempi di task corretti (*nome-corpo* consistenti) ed esempi di task non corretti (*nome-corpo* inconsistenti). Siccome Ansible è relativamente nuovo, non è semplice



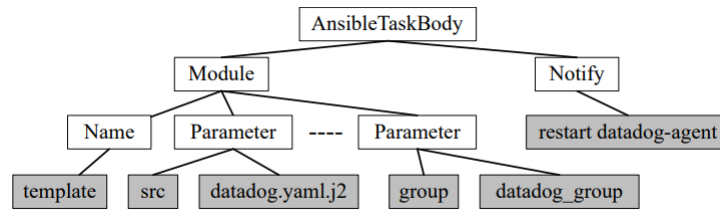
**Figura 3.2:** Panoramica sull’approccio di FindICI

trovare e collezionare un numero sufficiente di tasks inconsistenti da repositories reali. Borovits et al. [4], prendendo esempio da Pradel et al. [23] e da Li G. et al [19], hanno sviluppato un tool per le mutazioni di codice IaC, in modo da iniettare *buggy tasks* all’interno di un corpo di tasks Ansible apparentemente corretti. In particolare, le mutazioni prevedono semplicemente lo swap del corpo di un task con un altro. Per questa mutazione, Borovits et al. [4] hanno creato due tipi di varianti:

- una che riguarda lo stesso modulo, cioè vengono semplicemente scambiati i corpi di due moduli uguali (es. due moduli *service*);
- una riguardo tasks che usano moduli differenti; in questo caso, il corpo di un task di tipo *template* viene scambiato col corpo di un task di tipo *service*, ad esempio.

### Tokenizzazione dei Task Names e dei Task Bodies

Questo step converte le descrizioni degli Ansible tasks in un flusso di tokens che saranno poi utilizzati dagli algoritmi di apprendimento. Da un lato abbiamo i nomi dei tasks, che sono rappresentati generalmente da brevi descrizioni in linguaggio naturale. Dall’altro, invece, i loro corpi sono ben strutturati, come si può notare dalla Figura 2.1 del capitolo precedente. Tale struttura è preservata utilizzando l’ *abstract syntax tree* (AST) del task body per generare le sequenze di token, in modo da preservare la semantica del codice [4]. È stato creato, quindi, un modello AST per catturare le informazioni di un task body. La Figura 3.3 mostra un AST generato a partire da un task. I nodi dell’AST catturano la semantica, come i moduli e i loro parametri, quindi, il flusso di token generati da questo AST sarà *[AnsibleTaskBody, Module, Name, template, Parameter, src, datadog.yaml.j2, ..., Notify, restart, datadog-agent]* [4].



**Figura 3.3:** FindICI AST Model di un task

### Creazione delle rappresentazioni vettoriali

Per far apprendere gli algoritmi di apprendimento, le sequenze di token devono essere trasformate necessariamente in vettori. Quindi, Borovits et al. [4] hanno utilizzato tecniche di word embedding per generare i vettori relativi ai nomi e ai corpi dei tasks Ansible. Sono state create delle sequenze di token per ogni task e tali sequenze contengono i nomi e i corpi dei tasks. Successivamente, queste sequenze sono state date in input ai modelli di word embedding, i quali prendono un set di token di sequenze di stringhe e producono una mappa tra gli string token e i vettori numerici [22]. Le tecniche di word embedding incastrano i token con vettori numerici e posizionano semanticamente le parole simili in posizioni adiacenti. Come risultato, l'informazione semantica del testo è trasformata nel vettore numerico corrispondente. Prima di applicare le tecniche di word embedding, sono stati rimossi tutti i caratteri speciali dalle stringhe e, successivamente, sono stati uniti i token dei nomi dei task e i loro corpi. Ciò ha permesso a Borovits et al. [4] di costruire un singolo spazio vettoriale per ogni task, come già ottenuto in precedenti studi per la detection di inconsistenze *code-comment* [7; 23].

### Training e Tuning dei Prediction Models

Per effettuare il training di FindICI, sono stati utilizzati degli algoritmi di apprendimento per costruire un classificatore binario, affinché possa riuscire ad identificare i tasks in **consistenti** e **non consistenti**. I token generati dal word embedding risultano essere l'input del classificatore [4]. Prima di essere dati in pasto al classificatore, i token sono stati riempiti per conformarsi al meglio all'input di lunghezza fissa del classificatore; ciò è stato possibile aggiungendo vettori nulli alla fine delle sequenze di token [4].

### Identificazione delle inconsistenze

L'identificazione delle inconsistenze è un problema di classificazione binaria, in quanto i dati di test devono essere etichettati come *consistenti* o *inconsistenti*, ovvero rispettivamente classe positiva e negativa. Una volta che il classificatore è stato allenato con abbastanza dati, è possibile interrogarlo affinché analizzi un set di Ansible task ancora sconosciuti, classificandone le inconsistenze.

### Implementazione

Per analizzare e costruire gli AST dei tasks di Ansible, Borovits et al. [4] hanno sviluppato un tool in Python appositamente per effettuare tale operazione. Attraverso la libreria *NLTK*<sup>1</sup> sono stati tokenizzati i nomi dei tasks. Successivamente, tramite l'ausilio di *Word2vec*, *Doc2vec* e *FastText*, sono stati generati i vettori numerici per i tokens. Usando *Tensorflow*<sup>2</sup> e *Keras*<sup>3</sup>.

---

<sup>1</sup><http://www.nltk.org/>

<sup>2</sup><https://www.tensorflow.org>

<sup>3</sup><https://keras.io/>

Questo capitolo descrive il design dello studio empirico condotto. L'obiettivo del lavoro svolto è quello di andare a capire come il Machine Learning possa essere utilizzato per il rilevamento di inconsistenze linguistiche in IaC e quanto esso sia efficace, andando a consolidare i risultati dello studio precedentemente condotto da Borovits et al. [4].

#### 4.1 Definizione delle domande di ricerca

Come lo studio condotto da Borovits et al. [4], poniamo le seguenti domande di ricerca:

**RQ1.** In che modo il Machine Learning può rilevare inconsistenze linguistiche in IaC?

**RQ2.** In che misura le rappresentazioni in word embedding influenzano le prestazioni?

A queste domande è stata ristretta la rappresentazione word embedding a *Word2vec*, utilizzando *Continuous Bag of Words* e, successivamente, sono stati analizzati e valutati sei modelli di machine learning, ovvero XGBoost (XGB), Support Vector Machine (SVM), Random Forest (RF), Multi-Layer Perceptron (MLP), Convolutional Neural Networks (CNNs) e Long-Short Term Memory (LSTM). Per rispondere alla **RQ2**, sono state comparate diverse tecniche di word embedding per osservare la loro influenza sulle prestazioni del modello. Tutti gli esperimenti sono stati condotti su una macchina con una CPU AMD Ryzen 3700x, 32 GB di memoria RAM e una GPU NVIDIA RTX 3080.

## 4.2 Dataset

Per rispondere alle domande di ricerca, è stato utilizzato un corpus di dati reali di tasks Ansible minati da GitHub. È stato impiegato in particolare lo stesso dataset utilizzato (già tokenizzato) da Borovits et al. [4] nella ricerca che ha portato allo sviluppo di FindICI [4]. Per assicurare la qualità dei dati collezionati, Borovits et al. [4] hanno stabilito dei criteri di selezione adottati precedentemente da Rahman et al. [25] e Dalla Palma et al. [9].

**Criterio 1:** Almeno l'11% dei files appartenenti alla repository devono essere script IaC;

**Criterio 2:** La repository ha almeno 10 contributors;

**Criterio 3:** La repository deve avere almeno due commit al mese;

**Criterio 4:** La repository non è un fork.

Questi criteri sono già stati utilizzati in lavori precedenti per collezionare scripts IaC [9; 26]. In particolare, il criterio 1 rappresenta un cut-off che assicura la presenza di un numero sufficiente di scripts IaC tale da permettere un'analisi anche delle commit [4]. Lo studio condotto da Borovits et al. [4] e il presente considerano solo i 10 moduli più utilizzati, i quali contengono 10,396 tasks nel dataset collezionato. Questi tasks sono stati recuperati da 38 repositories di GitHub che rispondevano ai criteri indicati. Nella Tabella 4.1 è riportato il numero dei tasks suddivisi per modulo.

shell	command	set_fact	template	file	copy	gather_facts	service	debug	fail
2126	1702	1246	1198	1151	773	752	569	484	395

**Tabella 4.1:** Distribuzione dei tasks Ansible nel dataset

## 4.3 Workflow alternativo e generazione della mutazione

A differenza dello studio condotto da Borovits et al. [4] (che induce due tipi di mutazione al dataset), in questo lavoro si è scelto di indurre la mutazione di un singolo parametro del modulo analizzato, in modo da addestrare il modello alla ricerca di un'inconsistenza più *atomica*. Il parametro da mutare è scelto a caso tra quelli che possono assumere solo determinati valori definiti da Ansible. I possibili parametri candidati per la mutazione, nella documentazione Ansible sono caratterizzati dalla dicitura *Choices*, la quale indica i valori possibili del parametro. Inoltre, è stata effettuata un'ulteriore analisi e rielaborazione dei



tasks per eliminare parte delle informazioni non necessarie: siccome il focus principale è la mutazione parametro-valore, è stato tokenizzato solo il corpo del modulo con i suoi parametri. Nella Figura 4.1 è rappresentato il workflow alternativo seguito nella tesi. Si può notare che viene effettuata una mutazione delle descrizioni dei moduli dopo una prima tokenizzazione. Ciò perché risulta più vantaggioso "pulire" una lista di stringhe. Successivamente, viene effettuata una retokenizzazione dei moduli, sia quelli mutati che quelli considerati consistenti; questi token saranno rappresentati semplicemente dal nome del task e dal corpo del modulo, senza condizioni o altri argomenti non utili al rilevamento della mutazione dei parametri.

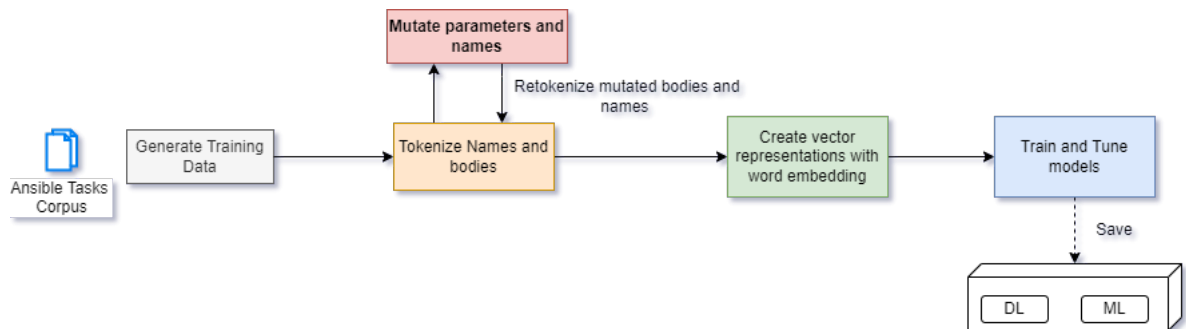
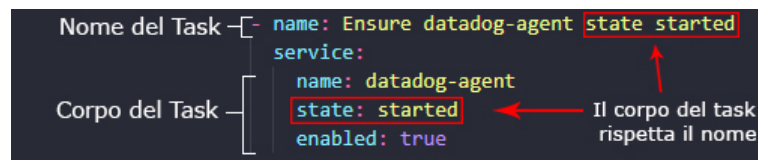
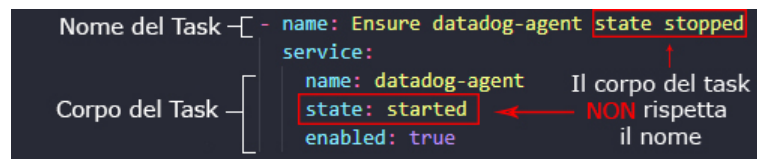


Figura 4.1: Workflow alternativo di FindICI



(a) Esempio di task consistente



(b) Esempio di task mutato e inconsistente

Figura 4.2: Task Ansible originale e task mutato

Nella Figura 4.2 è possibile notare la differenza tra un task originale e consistente con uno mutato. Nel task della in Figura 4.2 a, il corpo del task rispetta totalmente il nome che lo rappresenta. Nella Figura 4.2 b, invece, il nome del task indica che esso si assicura che il servizio *datadog-agent* ha uno stato uguale a *stopped*, quando invece risulta essere *started*. Ed è proprio questo il tipo di inconsistenza che si vuole rilevare.

Come già accennato, il metodo creato va a mutare un parametro del modulo tra quelli che presentano dei valori predefiniti e il suo nome, per aggiungere l'inconsistenza *nome-corpo* o meglio, in questo caso, *nome-parametro*. Il codice del *mutator* è interamente disponibile e documentato su GitHub<sup>1</sup>, con un esempio di utilizzo.

## 4.4 Selezione dei classificatori

Per rispondere alla prima *domanda di ricerca*, sono stati impiegati sei algoritmi di classificazione, ovvero *Random Forest (RF)* [16], *Support Vector Machine (SVM)* [8], *eXtreme Gradient Boosting (XGBoost)* [5], *Multi-Layer Perceptron (MLP)* [15], *Convolutional Neural Network (CNN)* [21] e *Long-Short Term Memory (LSTM)* [6]. Il loro utilizzo è stato ben pesato da Borovits et al. [4]. Più precisamente, è stato selezionato *Random Forest* per la sua resistenza al *noise* e alla bassa tendenza all'*overfitting*, così come *SVM*, il quale è stato scelto anche per la sua capacità di gestione dei dati non lineari [8]. *XGBoost*, invece, è meno influenzato dai datasets non bilanciati. Per quanto riguarda gli algoritmi basati su reti neurali, *MLP* è stato scelto per la sua semplicità, mentre *CNN* e *LSTM* per verificare se algoritmi più complessi forniscono performance migliori per il detect delle inconsistenze [4].

## 4.5 Selezione dei modelli

La selezione dei modelli è stata guidata da una *grid search* sui parametri dei modelli, attraverso una *k-fold cross-validation* stratificata. La *grid research* è un algoritmo di ricerca esaustiva, mentre lo *stratified k-fold cross-validation* è un metodo di convalida ampiamente utilizzato, il quale garantisce che ogni osservazione del dataset ha la possibilità di apparire nel training set e nel test set [4; 18]. Esso suddivide casualmente i dati in dieci *folds* di uguale dimensione, applicando un campionamento stratificato (i.e. ogni *fold* ha lo stesso numero di inconsistenze). Un solo *fold* viene utilizzato come test set, mentre gli altri sono utilizzati come training set. Tale processo viene ripetuto per dieci volte, utilizzando ogni volta un *fold* differente come test set. Le performance del modello sono successivamente riportate come una media delle dieci *run*. Questa metodologia non è stata applicata per le *CNNs* e *LSTM* né in questo studio né in quello precedente condotto da Borovits et al. [4], siccome è computazionalmente troppo dispendiosa [4]. Il dataset è stato splittato in tre set differenti

<sup>1</sup><https://github.com/woofz/ansible-tools/>

con una distribuzione equa delle inconsistenze: (i) il 60% è stato usato per il **training set**; (ii) un altro 20% è stato utilizzato per il **validation set**; (iii) il restante 20%, invece, per il **test set**.

## 4.6 Validazione dei modelli

I modelli sviluppati da Borovits et al. [4] saranno usati per predire le inconsistenze. Nel Machine Learning, ci sono quattro possibili risultati:

**True Positive (TP):** la classe attuale e quella predetta sono entrambi inconsistenti;

**False Negative (FN):** la classe attuale è inconsistente, ma la classe predetta è consistente;

**True Negative (TN):** la classe attuale e quella predetta sono entrambi consistenti;

**False Positive (FP):** la classe attuale è consistente, ma quella predetta è inconsistente.

Per valutare le performance dei modelli addestrati, sono state usate da Borovits et al. (e quindi anche in questo progetto) delle metriche comuni spesso usate nei problemi di classificazione binaria, ovvero: *accuracy*, *precision*, *recall*, *F1 Score*, *MCC* (*Matthew's Correlation Coefficient*), *AUC-ROC* (*Area Under the Receiver Operating Characteristic curve*).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 - score = \frac{precision \times recall}{precision + recall}$$

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

L'AUC misura l'intera area bidimensionale al di sotto dell'intero ROC (*receiver operating characteristic curve*, che traccia il tasso dei *True Positive* e quello dei *False Positive*). Un buon classificatore ha un valore di AUC vicino ad 1, mentre, uno scarso vicino allo 0 [4]. Per

confrontare le performances tra i classificatori e le tecniche di word embedding, sono state seguite le raccomandazioni di Demšvar et al. [12]. Nel lavoro di Borovits et al. [4] è stato applicato il Friedman test con un livello di significatività pari a 0.05 per rifiutare l'ipotesi nulla. Una volta stabilita una differenza statistica tra le prestazioni dei classificatori, è stata effettuata una *pairwise post hoc analysis* consigliata da Benavoli et al. [3], dove il comparatore di rank è sostituito dal test dei ranghi con segno di Wilcoxon con la *Holm's alpha correction*. Per comparare statisticamente le prestazioni di più classificatori e word embeddings sui moduli Ansible, i risultati sono stati tracciati sui diagrammi di *critical difference* (CD) [12] che permettono di visualizzare i risultati del test di Wilcoxon-Holm. In un diagramma CD, le posizioni delle entità analizzate (ad esempio tecniche di word embedding o metodi di classificazione) rappresentano la loro andatura media tra tutti i risultati delle osservazioni [4]. Due o più entità sono tra loro collegate da una linea orizzontale se non sono significativamente differenti in termini di metrica considerata; per eseguire l'analisi statistica e disegnare i diagrammi è stata usata la libreria sviluppata da Ismail Fawaz et al. [17].

## 4.7 Selezione delle tecniche di Word Embedding

Per rispondere alla **RQ2**, sono stati selezionati tre modelli di word embedding ampiamente utilizzati: *Word2Vec*, *Doc2vec* e *FastText*. Tali word embeddings sono largamente utilizzati nelle ricerche di ingegneria del software per l'apprendimento del codice sorgente e altri testi scritti in linguaggio naturale [13; 19; 20; 23]. *Word2vec* è una rete neurale a due layer che processa i testi creando rappresentazioni vettoriali a partire dalle parole. *Word2vec* per l'apprendimento può utilizzare sia *Continuous Bag of Words* (CBOW) sia *continuous skip-gram* per la rappresentazione delle parole. CBOW permette di predire una singola parola da un intervallo di dimensione fissa di un insieme di parole, mentre Skip-gram fa la predizione di più word contexts a partire da una singola parola di input. *Doc2vec* apprende, invece, rappresentazioni a partire da pezzi di testo di lunghezza variabile, come, ad esempio, frasi, paragrafi e documenti. È un'estensione di *Word2vec*, che considera l'ordine e la semantica delle parole racchiuse in un blocco di testo. *Doc2vec* può usare due tipi di architetture: *Distributed Bag of Words of Paragraph Vector* (PV-DBOW) e *Distributed Memory of Paragraph Vector* (PV-DM), che sono analoghe a Skip-gram e CBOW implementate da *Word2vec*. *FastText*, infine, è una sorta di *Word2vec* migliorato, in quanto prende in considerazione le parti delle parole (prefissi, radici e suffissi), spostando il training su dataset sempre più piccoli, generalizzando le parole sconosciute [4]. Utilizzando la soluzione proposta da Borovits et al. [4], sono state impiegate

le stesse impostazioni da loro utilizzate per ogni modello apprendimento di word embedding (Doc2Vec, Word2vec e FastText). In particolar modo:

- **context window size = 6**: tale parametro definisce il numero di parole che sono utilizzate per determinare il contesto di una parola; siccome i tasks Ansible sono brevi testi, Borovits et al. hanno optato per un valore uguale a 6;
- **vector size = 100**: è la dimensione del vettore che dev'essere appreso; siccome il corpus risulta essere abbastanza piccolo, è stata scelta una dimensione di 100 tokens, che è il valore di default usato dall'implementazione proposta da Borovits et al. con *gensim*.

## 4.8 Risultati

Questo paragrafo presenta i risultati ottenuti applicando gli algoritmi definiti. Le Tabelle dalla 4.2 alla 4.7 contengono le performance dei classificatori per la rilevazione delle inconsistenze per la *top ten* dei moduli Ansible con la mutazione indotta. Nella Figura 4.4 sono riportati i boxplots dei valori MCC, AUC ROC e Accuratezza dei classificatori considerati.

	shell	command	set_fact	template	file	gather_facts	copy	service	debug	fail
<b>AUC-ROC</b>	0.998	1	1	1	0.999	1	0.999	0.999	0.999	0.99
<b>MCC</b>	0.991	0.988	0.98	0.992	0.979	1	0.968	0.947	0.948	0.925
<b>Accuracy</b>	0.995	0.994	0.99	0.996	0.989	1	0.984	0.977	0.974	0.96
<b>F1-score</b>	0.995	0.994	0.99	0.996	0.989	1	0.984	0.97	0.975	0.964
<b>Precision</b>	1	0.988	0.992	0.991	0.979	1	0.975	0.97	0.97	0.941
<b>Recall</b>	0.99	1	0.988	1	1	1	0.994	0.97	0.98	0.988

**Tabella 4.2:** Risultati SVM per ogni modulo considerato

	shell	command	set_fact	template	file	gather_facts	copy	service	debug	fail
<b>AUC-ROC</b>	1	1	1	1	1	1	1	1	1	0.991
<b>MCC</b>	0.991	0.997	0.984	0.992	0.97	1	0.981	0.982	0.99	0.912
<b>Accuracy</b>	0.995	0.999	0.992	0.996	0.985	1	0.99	0.992	0.995	0.954
<b>F1-score</b>	0.995	0.998	0.992	0.996	0.985	1	0.99	0.99	0.995	0.958
<b>Precision</b>	0.998	0.997	0.996	0.996	0.979	1	0.981	0.99	0.99	0.94
<b>Recall</b>	0.993	1	0.988	0.996	0.991	1	1	0.99	1	0.975

**Tabella 4.3:** Risultati MLP per ogni modulo considerato

	shell	command	set_fact	template	file	gather_facts	copy	service	debug	fail
<b>AUC-ROC</b>	0.76	0.96	0.98	0.97	0.83	0.83	0.93	0.86	0.52	0.95
<b>MCC</b>	0.57	0.92	0.96	0.94	0.65	0.71	0.86	0.71	0.08	0.90
<b>Accuracy</b>	0.70	0.96	0.98	0.97	0.83	0.79	0.93	0.86	0.20	0.95
<b>F1-score</b>	0.80	0.96	0.98	0.96	0.81	0.87	0.92	0.85	0.69	0.94
<b>Precision</b>	0.69	0.99	0.97	0.99	0.84	0.77	0.98	0.83	0.55	1.00
<b>Recall</b>	0.96	0.93	0.99	0.94	0.79	1.00	0.87	0.87	0.92	0.89

**Tabella 4.4:** Risultati CNN per ogni modulo considerato

	shell	command	set_fact	template	file	gather_facts	copy	service	debug	fail
<b>AUC ROC</b>	0.995	0.990	0.990	1.000	0.983	1.000	0.999	0.940	1.000	0.924
<b>MCC</b>	0.937	0.909	0.880	0.967	0.857	0.993	0.942	0.689	0.979	0.650
<b>Accuracy</b>	0.970	0.957	0.937	0.984	0.928	0.997	0.972	0.851	0.990	0.827
<b>F1-score</b>	0.967	0.952	0.943	0.983	0.928	0.997	0.970	0.832	0.990	0.818
<b>Precision</b>	0.975	0.945	0.936	0.983	0.930	1.000	0.987	0.781	0.990	0.863
<b>Recall</b>	0.959	0.959	0.950	0.983	0.926	0.993	0.955	0.890	0.990	0.778

**Tabella 4.5:** Risultati RF per ogni modulo considerato

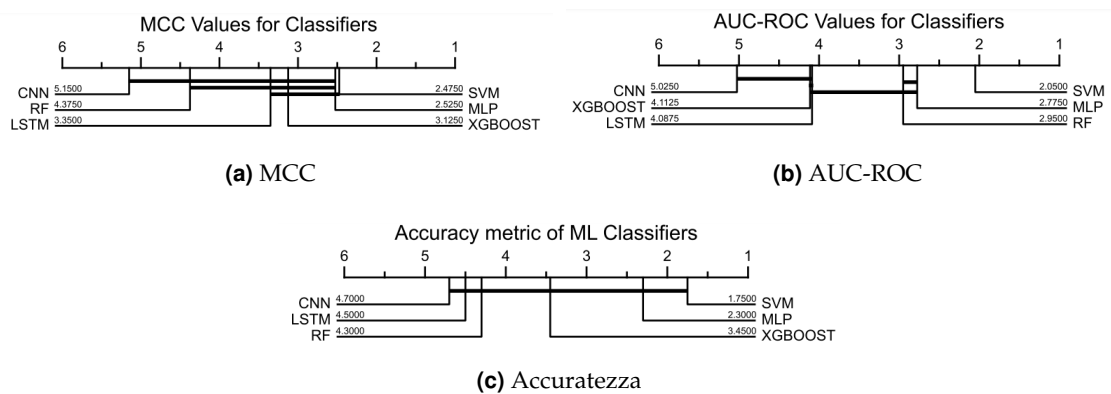
	shell	command	set_fact	template	file	gather_facts	copy	service	debug	fail
<b>AUC ROC</b>	0.979	0.987	0.977	0.986	0.952	0.990	0.971	0.903	0.979	0.860
<b>MCC</b>	0.958	0.974	0.952	0.971	0.905	0.980	0.942	0.805	0.959	0.723
<b>Accuracy</b>	0.980	0.987	0.975	0.986	0.952	0.990	0.971	0.913	0.979	0.851
<b>F1-score</b>	0.978	0.986	0.977	0.985	0.952	0.990	0.971	0.891	0.979	0.869
<b>Precision</b>	0.985	0.981	0.992	0.975	0.956	1.000	0.974	0.882	0.990	0.839
<b>Recall</b>	0.971	0.991	0.962	0.996	0.948	0.980	0.968	0.900	0.969	0.901

**Tabella 4.6:** Risultati XGBoost per ogni modulo considerato

	shell	command	set_fact	template	file	gather_facts	copy	service	debug	fail
<b>AUC ROC</b>	0.945	0.834	0.990	1.000	0.797	0.870	0.969	0.925	0.860	0.995
<b>MCC</b>	0.930	0.720	1.000	1.000	0.750	1.000	0.940	0.920	0.930	1.000
<b>Accuracy</b>	0.960	0.990	0.980	1.000	0.850	0.770	1.000	0.930	0.800	0.990
<b>F1-score</b>	0.940	0.790	1.000	1.000	0.780	1.000	0.950	0.920	0.910	1.000
<b>Precision</b>	0.951	0.858	0.988	0.998	0.811	0.857	0.974	0.925	0.845	0.994
<b>Recall</b>	0.949	0.855	0.988	0.998	0.810	0.865	0.972	0.925	0.845	0.994

**Tabella 4.7:** Risultati LSTM per ogni modulo considerato

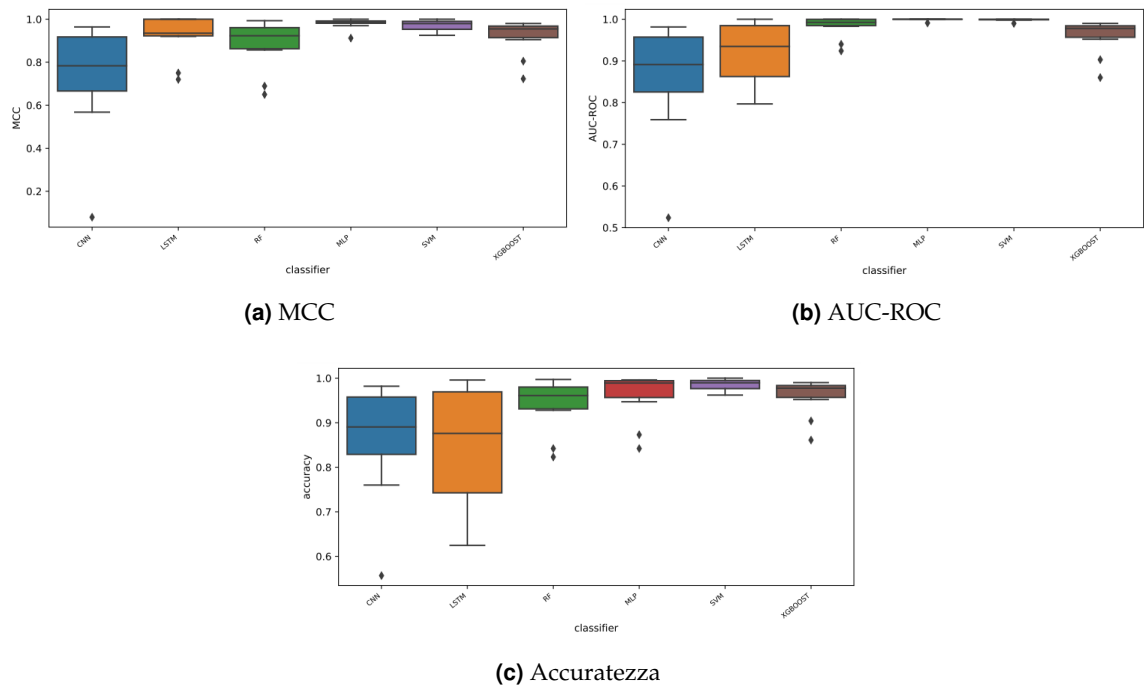
Nel complesso, come nei risultati osservati da Borovits et al. [4], l'algoritmo che ha ottenuto i risultati migliori è SVM. Questo classificatore ha un range di *accuracy* tra lo 0.96 e l'1 e AUC tra 0.99-1. SVM è capace di rilevare i tasks inconsistenti con un F1 score dallo 0.96 all'1, *recall* tra 0.97-1 e *precision* 0.94-1. È altresì vero che gli altri classificatori si avvicinano a tali performance. Tra i classificatori basati su reti neurali, MLP si conferma come miglior classificatore, mentre il peggiore risulta essere CNN, proprio come nello studio di Borovits et al. [4]. MLP fornisce *accuracy* tra 0.95 e 1, AUC da 0.99 e 1. Riesce ad identificare i tasks inconsistenti con un F1 score tra 0.95 e 1, *recall* da 0.97 e 1 e *precision* tra 0.94 e 1.



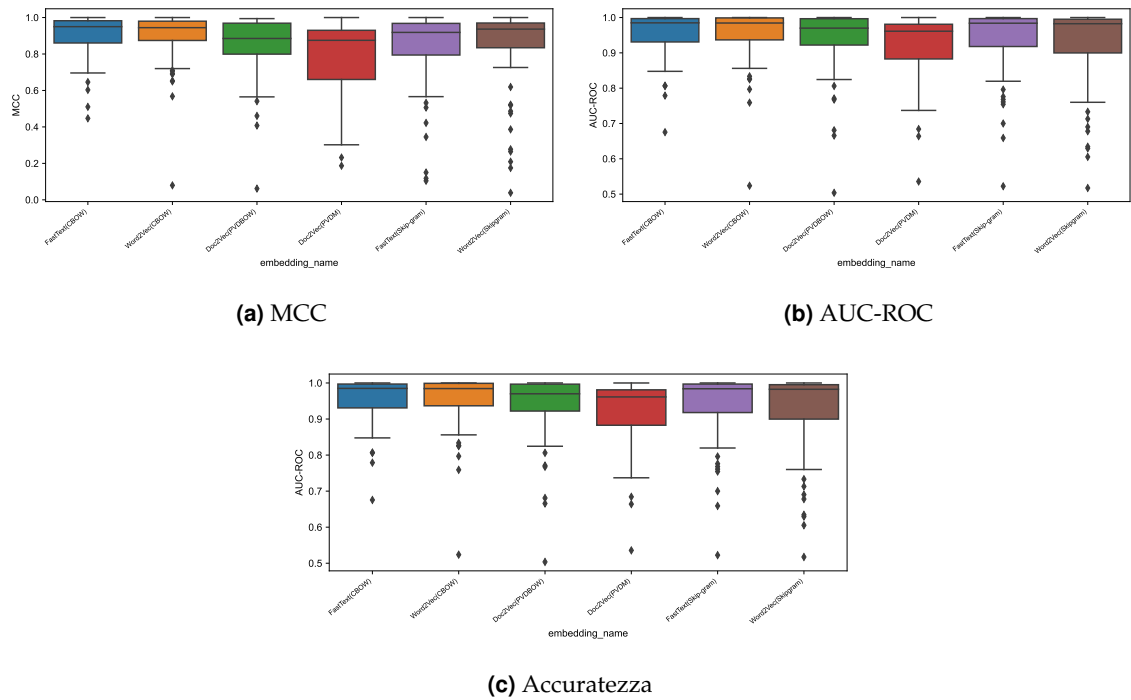
**Figura 4.3:** Critical Difference diagrams basati su test di Wilcoxon-Holm per il detect di differenze significative tra le performance sui vari moduli Ansible.

In Figura 4.3 sono riportati i risultati dell'analisi statistica condotta secondo il test di Wilcoxon-Holm. Seguendo le linee tracciate in grassetto, è possibile notare che non vi sono differenze significative tra i classificatori, sebbene SVM sia il più performante. Per quanto riguarda *MCC*, non ci sono differenze significative tra XGBoost e LSTM, così come per quanto riguarda l'*AUC-ROC*. Per l'*accuracy*, invece, RF e LSTM non presentano sostanziali differenze, così come LSTM e CNN. Con questi dati, proviamo a rispondere alla *domanda di ricerca 1*:

**RQ<sub>1</sub>:** gli algoritmi di machine learning possono essere utilizzati con successo per sviluppare dei classificatori in grado di rilevare inconsistenze linguistiche in IaC, confermando studi precedenti [4; 13]. Tuttavia, i modelli di deep learning dovrebbero essere ben analizzati prima di essere impiegati, in quanto risultano essere costosi da un punto di vista computazionale e, soprattutto, poco efficienti [4; 13].



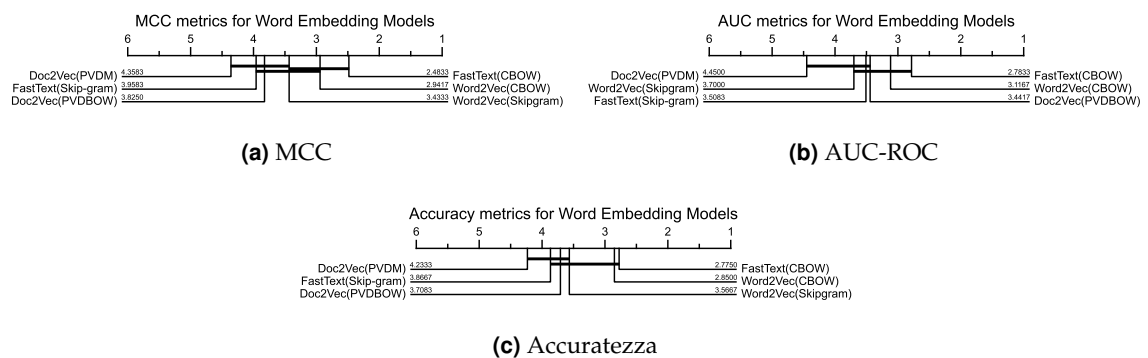
**Figura 4.4:** Boxplots che mostrano MCC, AUC-ROC e Accuratezza per ogni classificatore.



**Figura 4.5:** Risultati ottenuti in termini di MCC, AUC ROC e Accuratezza con Word Embedding

Per quanto riguarda la  $RQ_2$ , invece, si rimanda ai boxplots su MCC, AUC-ROC e accuratezza (Fig. 4.5) ottenuti applicando le differenti tecniche di word embedding ai classificatori.





**Figura 4.6:** Critical Difference diagram basato sul test Wilcoxon-Holm per le differenze tra i valori MCC, AUC ROC, Accuratezza ottenuti applicando le tecniche di Word Embedding proposte

I risultati suddivisi per algoritmo e per tecnica di word embedding sono presenti sulla repository<sup>2</sup> del progetto, che risulta essere un fork di quello di Borovits et al. [4] con modifiche al metodo di mutazione e rielaborazioni sulla tokenizzazione. Come è possibile notare, tutti i modelli presentano delle buone performance per le metriche considerate. Tuttavia, i modelli basati su *FastText* e *Word2Vec* risultano essere quelli con prestazioni superiori, insieme alle loro varianti basate su Skip-gram. Il modello *Doc2Vec* basato su *PV-DBOW*, invece, risulta essere più performante rispetto a quello basato di *PV-DM*. I risultati sono confermati nell'analisi statistica, rappresentata in Figura 4.6. I risultati evidenziano una certa superiorità di *FastText* e *Word2Vec*. Riguardo le metriche di accuratezza, non ci sono differenze statisticamente rilevanti tra *FastText* (CBOW) e *Word2Vec*. Il modello *Doc2Vec* (PV-DBOW) sembra guadagnare di poco su *FastText* basato su Skip-gram. Per quanto riguarda *Doc2vec* (PV-DM), esso risulta essere il meno performante. Possiamo fornire una risposta anche alla domanda di ricerca 2:

**RQ<sub>2</sub>:** I modelli addestrati con *Word2vec* e *FastText* risultano essere più performanti rispetto a quelli allenati utilizzando *Doc2vec*. Inoltre, i modelli trainati con *Continuous Bag of Words* sembrano essere superiori rispetto a quelli allenati con *Skip-gram*, anche se le differenze non sono statisticamente significative.

## 4.9 Minacce alla validità

Avendo utilizzato come base lo studio condotto da Borovits et al. [4], la tesi presenta in linea di massima le stesse minacce interne, esterne e di costrutto.

<sup>2</sup><https://github.com/woofz/FindICI>

**Minacce alla validità di costruito.** Le repositories selezionate sono le stesse scelte da Borovits et al., quindi presentano gli stessi limiti. Esse potrebbero non essere rilevanti per il problema considerato. Per ridurre tale rischio, Borovits et al. [4] le hanno selezionate in base a criteri utilizzati in studi precedenti su *smell detection* di IaC. Un'altra minaccia riguarda il metodo utilizzato per generare la mutazione, il quale potrebbe non rappresentare a pieno i casi di Ansible tasks inconsistenti del mondo reale. Nel dataset utilizzato, i tasks consistenti e inconsistenti sono presenti in ugual numero e ciò non rispecchia una situazione reale, nella quale è quasi impossibile che siano presenti in egual numero. Nonostante ciò, Borovits et al. [4] si sono basati su studi già consolidati [19; 23], che hanno mutato il dataset con successo, ottenendo ottimi risultati.

**Minacce alla validità interna.** La scelta delle caratteristiche utilizzate per il training dei classificatori potrebbe influenzare la detection degli antipattern linguistici. Tale minaccia è stata ridotta addestrando il modello utilizzando diverse caratteristiche (trasformando ogni task in uno spazio vettoriale di parole) estratte da oltre diecimila tasks di Ansible. La qualità del codice utilizzato per il training stabilisce anche il livello di apprendimento del classificatore. I typos, le abbreviazioni e le convenzioni di naming sono tutte caratteristiche del codice infrastrutturale che vanno ad incidere sul training del classificatore. Per mitigare questo aspetto, sono state utilizzate tecniche avanzate di NLP [4].

**Minacce alla validità esterna.** I risultati sono stati raggiunti considerando solo un sottoinsieme di moduli Ansible. Di conseguenza, i modelli impiegati non potranno essere utilizzati per altri modulo al di fuori di quelli usati per l'addestramento (ovvero i 10 moduli più utilizzati). È, comunque, in programma un'estensione delle funzionalità dei classificatori; infatti è stata sviluppata una funzione integrativa al *mutator*, la quale, dato un modulo Ansible, recupera dalla documentazione i suoi parametri con i valori che possono assumere.

In questa tesi è stato valutato l'utilizzo del machine learning e del deep learning sull'attività di rilevazione di inconsistenze linguistiche in Infrastructure as Code. L'obiettivo principale, oltre alla validazione del Machine Learning e del Deep Learning, è quello di consolidare lo studio precedentemente condotto da Borovits et al., estendendo l'utilizzo del tool FindICI da loro sviluppato ad un antipattern linguistico più specifico. È stato utilizzato il dataset dello studio di Borovits et al. [4] come base di partenza. Ad esso è stata applicata una mutazione con un metodo sviluppato ad-hoc per lo studio. Tale mutazione ha generato l'inconsistenza *nome-corpo* (più precisamente *nome-parametro*) per ogni Ansible task, andando a mutare un parametro tra quelli disponibili e il nome del task stesso. Poi, dopo aver applicato le tecniche di Word Embedding dello studio precedentemente effettuato da Borovits et al. [4], sono stati allenati i modelli di machine learning e deep learning, affinché i classificatori fossero in grado di riconoscere l'inconsistenza iniettata. Per rispondere alla prima domanda di ricerca posta nel Capitolo 4, è stata fissata l'analisi dei risultati alla rappresentazione Word Embedding data da Word2Vec. Dei sei algoritmi utilizzati, per la rappresentazione Word2Vec, la *Support Vector Machine* è risultata la più performante in termini di accuratezza, MCC e AUC ROC. Le performance offerte dal *Multi-Layer Perceptron* non si allontanano di molto da quelle di SVM, infatti la differenza non è statisticamente significativa, come riportato in Figura 4.3 del Capitolo 4. Per la seconda domanda di ricerca, invece, sono stati selezionati e comparati tre modelli di word embedding: Word2Vec, Doc2Vec e FastText, in quanto largamente utilizzati per l'apprendimento di codice in linguaggio naturale. Dall'analisi statistica emerge

che FastText e Word2Vec risultano essere superiori, come riportato dai diagrammi di Critical Difference nella Figura 4.6. L'analisi finale dei risultati fa risaltare quanto il Machine Learning sia valido per la ricerca di inconsistenze in Infrastructure as Code. Nonostante ciò, i modelli di deep learning devono essere analizzati con attenzione prima di essere utilizzati, in quanto risultano essere computazionalmente costosi.

**Sviluppi futuri.** Lo studio ha messo in luce aspetti interessanti sui quali sarà possibile concentrare le ricerche future. Anche se studi precedenti hanno utilizzato con successo la metodologia dell'induzione delle mutazioni ai fini dello studio [4; 19; 23], il dataset mutato potrebbe non rispecchiare casi reali di inconsistenze e, applicando il modello proposto, i risultati potrebbero essere discordanti con quelli raggiunti in questo studio. Anche per quanto riguarda il numero di moduli "coperti" potrebbero esserci risultati inattesi. Il modello fornito da questo studio è addestrato sui dieci moduli più utilizzati di Ansible; quindi, utilizzando dati reali con moduli differenti da quelli analizzati, i risultati potrebbero non essere soddisfacenti. Ciò significa che potrebbe essere necessario condurre un'analisi qualitativa del modello, applicandolo a dati reali. Un altro studio interessante potrebbe riguardare la comparazione dei risultati del modello proposto con i risultati di un tool sviluppato mediante l'utilizzo dei classici approcci dell'ingegneria del software, senza ausilio del Machine Learning.

**Esperienza al JADS.** Riservo quest'ultima parte della tesi all'esperienza che ho avuto modo di fare presso l'organizzazione *Jheronimus Academy of Data Science*. Innanzitutto, consiglio vivamente agli studenti di partecipare al programma Erasmus, nonostante possa spaventare l'idea di doversi adattare a nuove realtà. L'Erasmus è un'esperienza unica: si ha modo di sperimentare contesti internazionali diversi dai nostri, che, in qualche modo, portano ad una crescita personale non solo da un punto di vista formativo-professionale, ma anche da un punto di vista umano. Oltre a ciò, mi ritengo fortunato per essermi ritrovato al Jheronimus Academy of Data Science, un ambiente colmo di persone stimolanti che mi hanno acceso la curiosità di sapere, la voglia e la consapevolezza di poter dare sempre il massimo grazie alla loro preparazione e ad una gestione ottima del team. Inoltre, al mio rientro in Italia, la distanza non ha raffreddato i rapporti che si sono creati, in quanto l'umanità delle persone incontrate si è consolidata; in particolar modo con i professori che ho scelto come relatori della tesi, i quali mi hanno sopportato e supportato nella stesura effettiva del lavoro svolto presso il JADS.

---

## Ringraziamenti

---

Questo spazio è dedicato interamente alle persone che hanno lasciato qualcosa in me e che occuperanno sempre un posto nel mio cuore.

Un ringraziamento in particolare va ai miei relatori, i professori Fabio Palomba e Dario Di Nucci che mi hanno seguito con la loro disponibilità e infinita pazienza nel sopportarmi. Senza di loro, l'esperienza nei Paesi Bassi non si sarebbe concretizzata. Li considero come modelli da seguire per la loro dedizione al lavoro e alla ricerca, per la professionalità e la preparazione; sono un valore aggiunto e inestimabile dell'Università degli Studi di Salerno. Mi ritengo molto fortunato per averli incrociati durante il percorso della vita, grazie di tutto. Ringrazio tutto il JADE Lab del JADS, in particolar modo il professor Damian A. Tamburri, che, con la sua contagiosa energia, si è prodigato per darmi supporto e per permettere la mia partenza, Stefano Dalla Palma, Daniel De Pascale e Mirella Sangiovanni per avermi fortemente sostenuto e, soprattutto, per avermi fatto sentire a casa. Siete delle persone uniche e straordinarie, spero che le nostre strade si incrocino di nuovo in futuro.

Ringrazio i miei genitori, Cosimo e Sueva, per il loro sostegno costante e i loro insegnamenti senza i quali non sarei ciò che sono oggi. Senza di voi tutto ciò non sarebbe stato possibile e le parole non possono esprimere quanto io sia grato per tutti i sacrifici che avete fatto per me. Questa laurea è anche vostra e spero che oggi possiate essere felici.

Ringrazio mia sorella Lina, che continua a prendersi cura di me e a proteggermi, sostenendomi nei momenti duri e dandomi una scossa in quelli di confusione, mio cognato Fabio, sempre disponibile per qualsiasi circostanza. Ringrazio anche quei due pazzi dei miei nipoti, Lorenzo e Danilo, che mi hanno fatto scoprire un nuovo tipo di amore.

Un ringraziamento particolare va alla mia fidanzata, Federica, che ha sempre creduto nelle mie capacità e mi ha spronato a dare sempre il meglio. Tutto è partito da una tua idea, detta quasi per scherzo: partecipare all'Erasmus. Posso dire che sei quasi un coautore di questo

lavoro. Grazie per la tua pazienza, per la forza che mi hai trasmesso in tutti questi anni per portare a termine il percorso universitario e non solo. Se ce l'ho fatta è anche merito tuo. Lo sai.

Un ringraziamento va soprattutto a "Cumbà" Biagio e a nonna "Wae" Caterina, che risplendono nelle luci della nostra piazzetta. Sono sicuro che oggi siete felici e orgogliosi di me. Mancate più dell'aria.

Un grazie alla MDC che mi ha accompagnato fin dal principio. Siamo davvero fortunati di poter contare gli uni sugli altri; il legame nato fra tutti noi è inscindibile.

Ringrazio i miei amici di sempre, Emanuele e Vincenzo, per tutte le volte che hanno sopportato i miei sfoghi e per avermi aiutato a superare situazioni difficili.

Un ringraziamento generale va a tutti i miei parenti che non posso elencare per motivi di spazio. Li ringrazio per il loro supporto e l'appoggio ricevuto.

- [1] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016. (Citato a pagina 19)
- [2] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196. IEEE, 2013. (Citato alle pagine 10, 11 e 19)
- [3] Alessio Benavoli, Giorgio Corani, and Francesca Mangili. Should we really use post-hoc tests based on mean-ranks? *The Journal of Machine Learning Research*, 17(1):152–161, 2016. (Citato a pagina 29)
- [4] N Borovits, I Kumara, D Di Nucci, K Parvathy, F Palomba, D. A. Tamburri, S Dalla Palma, and W. J. van den Heuvel. Findici: Using machine-learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. 2022. (Citato alle pagine i, 1, 2, 14, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 32, 34, 35, 36 e 37)
- [5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016. (Citato a pagina 27)
- [6] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016. (Citato a pagina 27)

- [7] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, 2018. (Citato a pagina 22)
- [8] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. (Citato a pagina 27)
- [9] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A Tamburri. Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Transactions on Software Engineering*, 48(6):2086–2104, 2021. (Citato alle pagine 19 e 25)
- [10] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, 170:110726, 2020. (Citato a pagina 19)
- [11] Stefano Dalla Palma, Dario Di Nucci, and Damian A Tamburri. Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible. *SoftwareX*, 12:100633, 2020. (Citato a pagina 19)
- [12] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine learning research*, 7:1–30, 2006. (Citato a pagina 29)
- [13] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol. Keep it simple: Is deep learning good for linguistic smell detection? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611. IEEE, 2018. (Citato alle pagine 29 e 32)
- [14] Michele Guerriero, Martin Garriga, Damian A Tamburri, and Fabio Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589. IEEE, 2019. (Citato a pagina 19)
- [15] Simon Haykin and Richard Lippmann. Neural networks, a comprehensive foundation. *International journal of neural systems*, 5(4):363–364, 1994. (Citato a pagina 27)
- [16] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995. (Citato a pagina 27)



- [17] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data mining and knowledge discovery*, 33(4):917–963, 2019. (Citato a pagina 29)
- [18] G James, D Witten, T Hastie, and R Tibshirani. An introduction to statistical learning, vol 112 springer. *New York*, 2013. (Citato a pagina 27)
- [19] Guangjie Li, Hui Liu, Jiahao Jin, and Qasim Umer. Deep learning based identification of suspicious return statements. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 480–491. IEEE, 2020. (Citato alle pagine 21, 29, 35 e 37)
- [20] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1–12. IEEE, 2019. (Citato a pagina 29)
- [21] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*, 16(5-6):555–559, 2003. (Citato a pagina 27)
- [22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013. (Citato a pagina 22)
- [23] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018. (Citato alle pagine 21, 22, 29, 35 e 37)
- [24] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(1):1–31, 2021. (Citato alle pagine 14 e 17)
- [25] Akond Rahman and Laurie Williams. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*, pages 34–45. IEEE, 2018. (Citato a pagina 25)
- [26] Akond Rahman and Laurie Williams. Source code properties of defective infrastructure as code scripts. *Information and Software Technology*, 112:148–163, 2019. (Citato alle pagine 19 e 25)

- 
- [27] Gerald Schermann, Sali Zumberi, and Jurgen Cito. Structured information on state and evolution of dockerfiles on github. In *Proceedings of the 15th international conference on mining software repositories*, pages 26–29, 2018. (Citato a pagina 19)
- [28] Julian Schwarz, Andreas Steffens, and Horst Lichter. Code smells in infrastructure as code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228. IEEE, 2018. (Citato alle pagine 10 e 14)
- [29] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 189–200. IEEE, 2016. (Citato alle pagine 10, 14, 15, 16 e 19)