Plagiarism detection for python source codes

Giulio Taralli and Ismaila Toure

Università Degli Studi di Torino

Abstract. Questo documento propone un algoritmo per il rilevamento del plagio nei codici scritti in linguaggio Python, focalizzandosi su due aspetti chiave: la similarità semantica e la struttura del codice. Per l'aspetto semantico, utilizziamo la cosine similarity per confrontare i due codici sorgenti. Per l'aspetto strutturale, analizziamo gli Abstract Syntax Tree (AST) per identificare somiglianze nella struttura dei codici sorgente. Combinando questi due approcci l'algoritmo fornisce una stima della probabilità di plagio.

Keywords: NLP · plagiarism detection · python

1 Introduzione

Il rilevamento del plagio nel codice sorgente è un compito complesso. A differenza di altri contesti in cui il plagio può essere facilmente identificato, nella programmazione esistono situazioni in cui soluzioni simili o identiche possono emergere indipendentemente senza che vi sia stata alcuna forma di copiatura. Questo è particolarmente vero in casi come l'implementazione di algoritmi semplici o noti, ad esempio il calcolo del fattoriale, dove il numero di approcci validi è limitato e, di conseguenza, diversi programmatori possono giungere a soluzioni molto simili. Considerando questi fattori è evidente che un approccio superficiale, basato solo sul confronto diretto del codice, risulterebbe inefficace o addirittura fuorviante.

2 Scelte implementative

L'algoritmo proposto analizza il codice sorgente sotto due prospettive principali. La prima è l'aspetto semantico, per il quale viene utilizzata la cosine similarity, una tecnica che consente di misurare la somiglianza tra i vettori semantici del codice. Questo approccio permette di rilevare similarità di significato anche in presenza di riformulazione del codice. La cosine similarity rimane pertinente anche nel contesto dei linguaggi di programmazione in quanto quest'ultimi sono classificati come linguaggi liberi dal contesto.

La seconda prospettiva è l'analisi simbolica e strutturale, che sfrutta gli Abstract Syntax Tree (AST) per esaminare la struttura profonda del codice sorgente. Gli AST evidenziano la struttura logica e l'aspetto simbolico del programma, indipendentemente dalle variazioni sintattiche.
Combinando queste due tecniche, l'algoritmo è in grado di fornire una stima accurata della probabilità di plagio.

2.1 Analisi semantica

Questa fase si concentra sull'interpretazione del significato semantico del codice per rilevare somiglianze che potrebbero indicare un plagio. Una tecnica efficace per misurare tale somiglianza è la similarità del coseno.

La similarità del coseno è una misura che quantifica la somiglianza tra due vettori (rappresentazioni numeriche del codice) nello spazio n-dimensionale, utilizzando il coseno dell'angolo tra di essi:

$$\text{similarità del coseno} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

$$\cos(\theta) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \cdot \sqrt{\sum_{i=1}^{n} b_i^2}}$$

La similarità del coseno restituisce dei valori compresi tra [-1, 1] che vanno interpretati nel seguente modo:

- 1: I vettori puntano nella stessa direzione, cioè i due codici sono identici
- 0: I vettori sono ortogonali, cioè i codici non sono correlati semanticamente.
- -1: I vettori puntano in direzioni opposte, cioè i codici sono completamente diversi nel significato.

In maniera generale, la similarità del coseno dovrebbe sempre restituire dei valori compresi tra [0, 1]. Più il valore è vicino a 1, più i due documenti sono semanticamente simili e questo potrebbe indicare un eventuale plagio. Per l'implementazione si è utilizzata la funzione cosine_similarity della libreria scikit-learn.

```
vectorizer = TfidfVectorizer(stop_words=COMMON_KEYWORDS['python'], max_features=1000).fit_transform([codeA, codeB])
    vectors = vectorizer.toarray()
    similarity = cosine_similarity(vectors)[0, 1]
```

Fig. 1. utilizzo della funzione di scikit-learn

2.2 Analisi strutturale e simbolica

L'analisi strutturale e simbolica del codice sorgente è fondamentale per rilevare somiglianze e differenze a livello di struttura e simbolismo. Per questo scopo, utilizziamo gli Abstract Syntax Tree (AST). Gli AST sono strutture ad albero che rappresentano la struttura sintattica del codice sorgente in modo astratto. Ogni nodo dell'albero corrisponde a una costruzione sintattica, come una dichiarazione, un'espressione o un'operazione. L'uso degli AST permette di analizzare il codice a un livello di dettaglio superiore rispetto alla sola valutazione semantica.

Per rilevare le eventuali differenze, esaminiamo ricorsivamente i nodi dei due AST confrontandoli per calcolare una probabilità di plagio per ogni sottoalbero. L'algoritmo utilizzato è il seguente:

```python
def compare_ast_nodes(nodeA, nodeB):
    if type(nodeA) != type(nodeB):
        return 0.1

    score = 1.0
    for field in nodeA._fields:
        valueA = getattr(nodeA, field, None)
        valueB = getattr(nodeB, field, None)

        if isinstance(valueA, list) and isinstance(valueB, list):
            listSimilarity = compare_ast_lists(valueA, valueB)
            score *= listSimilarity
        elif isinstance(valueA, ast.AST) and isinstance(valueB, ast.AST):
            nodeSimilarity = compare_ast_nodes(valueA, valueB)
            score *= nodeSimilarity
        else:
            if valueA != valueB:
                score *= 0.9

    return score
```

Fig. 2. funzione ricorsiva che visita i nodi dell'AST

```python
def compare_ast_lists(childListA, childListB):
    if len(childListA) != len(childListB):
        return 0.3
    if len(childListA) == 0 and len(childListB) == 0:
        return 1

    scores = [compare_ast_nodes(n1, n2) for n1, n2 in zip(childListA, childListB)]
    return sum(scores) / len(scores)
```

Fig. 3. funzione di calcolo degli scores dei sottoalberi

L'algoritmo parte dall'assunto ottimista che ci sia plagio, impostando la probabilità (score) a uno. Ogni volta che rileva delle differenze, applica delle penalità

per ridurre la probabilità. In base alla situazione, delle penalità diverse vengono applicate allo score per ridurre la probabilità di plagio. Ad esempio, nel caso in cui due nodi dell'albero sono diversi, viene applicato una lieve penalità moltiplicando lo score per 0,9. Invece se le lunghezze dei due sottoalberi sono completamente diverse, si applica una penalità più forte in quanto si ha la certezza di avere due strutture diverse.

2.3 Calcolo finale della probabilità

Per determinare la probabilità finale di plagio tra due codici sorgenti, combiniamo le misure di similarità ottenute dalle analisi semantica e strutturale. La probabilità finale è calcolata attraverso una somma pesata delle similarità, utilizzando i seguenti parametri:

$$\text{Probabilità Finale} = (\alpha \times \text{Cosine\_similarity}) + (\text{AST\_similarity} \times \beta)$$

Dove:

- **alpha**: Il peso attribuito alla similarità del coseno.
- **beta**: Il peso attribuito alla similarità calcolata tramite il confronto degli Abstract Syntax Tree (AST).

Per il calcolo della probabilità finale, abbiamo deciso di attribuire un peso di 0,7 alla similarità degli AST e un peso di 0,3 alla similarità del coseno. Questo approccio riflette la nostra valutazione secondo cui le somiglianze strutturali sono più indicative di un possibile plagio rispetto alle somiglianze semantiche. Le differenze strutturali nel codice possono rivelare divergenze profonde nella logica e nell'organizzazione del programma, mentre le somiglianze semantiche potrebbero essere più comuni e meno indicative di plagio.

3 Risultati e limiti

In questa sezione verranno analizzati cinque principali esempi con i relativi risultati che fanno esaltare i pregi e i limiti di questo approccio.

3.1 Algoritmo del fattoriale

Questo esempio è un classico esercizio didattico per imparare il calcolo fattoriale e il concetto della ricorsione.

```python
def calcola_fattoriale(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * calcola_fattoriale(n - 1)

numero = int(input("Inserisci un numero intero non negativo: "))
risultato = calcola_fattoriale(numero)
print(f"Il fattoriale di {numero} è: {risultato}")
```

Fig. 4. prima implementazione del fattoriale

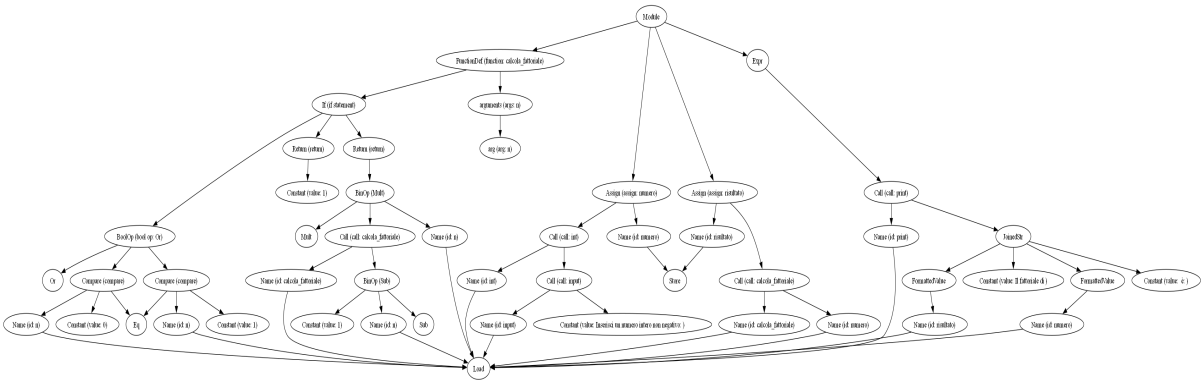

Fig. 5. Abstact Syntax Tree della prima implementazione

```python
def factorial(num):
    if num == 0 or num == 1:
        return 1
    else:
        return num * factorial(num - 1)

n = int(input("Inserisci un numero intero non negativo: "))
result = factorial(n)
print(f"Il fattoriale di {n} è: {result}")
```

Fig. 6. seconda implementazione del fattoriale

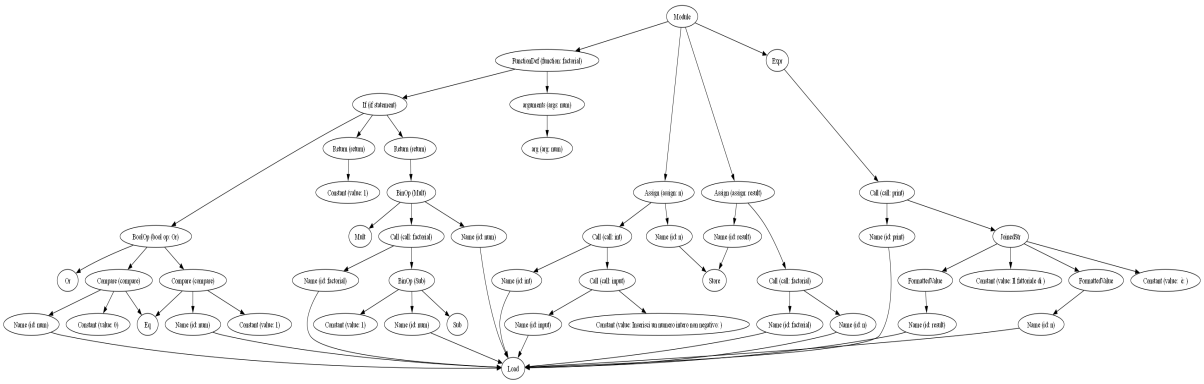

Fig. 7. Abstact Syntax Tree della seconda implementazione

In questi due codici, le uniche differenze sostanziali sono al livello simbolico: I nomi delle funzioni e delle variabili cambiano leggermente, ma la struttura generale è uguale.
In un contesto didattico, difficilmente questo può essere considerato plagio in quanto è un algoritmo noto con un approcio noto, quindi difficilmente si possono avere dei codici diversi l'uno dall'altro. In questo caso, riconoscere il contesto aiuta.

In questo caso l'algoritmo produce questo risultato:

```
Results:
Similarity prob: 61.08%
    Cosine similarity: 22.15%
    AST similarity probability: 77.76%

Comments:
Slight probability of plagiarism. Human control is recommended.
    The patters/simbologies of the two codes are too similar.
```

Fig. 8. Probabilità di plagio per l'algoritmo fattoriale

La percentuale di plagio risulta essere quindi un 61.08%, questo dovuto al fatto che nei due codici si è usata una semantica diversa che nonostante il peso più basso, tende ad abbassare di circa 16% la probabilità di plagio finale.
Tale percentuale non è cosi alta da assicurarsi la certezza di un plagio, molto dipende dal contesto. In questo caso, si invita ad un controllo umano.

3.2 Algoritmo DFS

In questo esempio i due codici sono completamente diversi, nonostante qualche nome di variabili identici, le due strutture sono completamente diverse: una ricorsiva e l'altra iterativa.
In questo caso l'algoritmo produce questo risultato:

```
Results:
Similarity prob: 49.89%
    Cosine similarity: 54.59%
    AST similarity probability: 47.88%

Comments:
Small probability of plagiarism.
```

Fig. 9. Probabilità di plagio per l'algoritmo DFS

In questo caso la percentuale risulta poco inferiore al 50% suggerendo una scarsa probabilità di plagio.

3.3 Algoritmo OOP Negozio

In questo esempio di programmazione orientata agli oggetti, i due codici sono simili: l'unica cosa che cambia è il pattern/l'ordine delle funzioni dell'oggetto

negozio. In questo caso è chiara la presenza di plagio.
In questo caso l'algoritmo produce questo risultato:

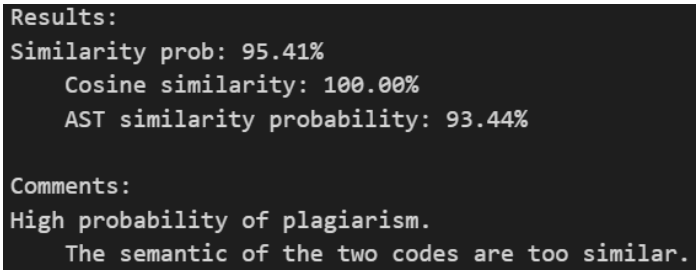

Fig. 10. Probabilità di plagio per l'algoritmo Negozio

3.4 Algoritmo Media somma

In questo esempio, il programma prende in input un array di numeri e calcola le seguenti operazioni: media degli elementi, somma degli elemente, calcolo del minimo e il calcolo del massimo.
Nonostante la somiglianza semantica tra i due codici, la struttura di quest'ultimi è completamente diversa in quanto uno utilizza una serie di funzioni per calcolarsi i vari valori per poi restituire un oggetto alla fine, mentre l'altro sfrutta la OOP, definendo una classe e calcolando i vari valori tramite dei metodi. In questo caso l'algoritmo produce questo risultato:

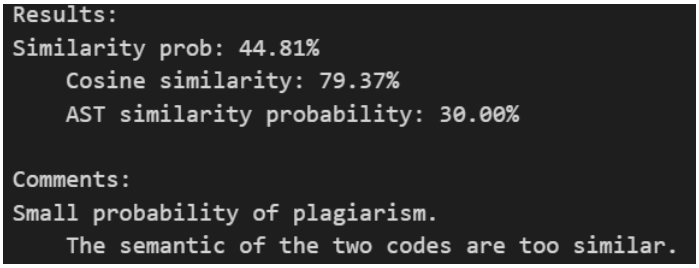

Fig. 11. Probabilità di plagio per l'algoritmo Media Somma

Nonostante l'alta percentuale della similarità del coseno, quella dedotta da AST risulta essere molto bassa. Avendo deciso di dare più peso alla probabilità di AST, la probabilità finale risulta essere sotto al 50%.

3.5 Considerazioni finali, pregi e limiti

L'algoritmo implementato si comporta tendenzialmente nella maniera corretta nella maggior parte dei casi riconoscendo evidenti plagi o affermando l'assenza di plagio tra i due codici.
Restituendo una percentuale, diamo una sfumatura maggiore nell'interpretazione finale dell'utente definendo un comportamento specifico in base al range di probabilità che l'algoritmo restituisce:

- tra 0% e 50%: Si può essere abbastanza certi dell'assenza di plagio.
- tra 50% e 70%: In questo caso, l'algoritmo ha rilevato delle parti di codici molto simili ma non si è sicuri di avere un caso conclamato di plagio. È necessario un controllo umano per valutare la situazione in base al contesto.
- tra 70% e 100%: Si può essere abbastanza certi della presenza di un plagio.

Nonostante questo, ci sono dei casi in cui la percentuale restituita dall'algoritmo risulta essere imprecisa, come ad esempio nel caso di codici molto corti e implementazioni di algoritmi noti (come l'agoritmo di Dijkstra).

Invece, codici lunghi facenti parte di implementazioni più ampie, anche se hanno variabili simili oppure algoritmi noti al loro interno, l'algoritmo si comporta molto bene, restituendo al più una probabilità che suggerisce un controllo umano.

4 Future implementazioni

Una volta discussi i risultati e i limiti delle tecniche descritte in precedenza, possiamo considerare le seguenti strategie che potrebbero migliorare la precisione del nostro algoritmo:

- **Tecniche di machine learning e deep learning**: Queste tecniche permettono di identificare schemi complessi e correlazioni nei dati tramite modelli matematici che potrebbero sfuggire agli approcci tradizionali descritti in questo progetto.
  Si potrebbe usare quindi un mdello per effettuare delle classificazioni utlizzando un dataset di codici etichettati come plagiati/non plagiati per l'allenamento.
  I modelli di ML/DL che ben si adattano a questo task sono le Support Vector Machine, le Random Forest e le reti neurali profonde.

- **Analisi temporale delle modifiche**: Questa tecnica, oltre a permette l'identificazione di una probabilità di plagio, permette di capire quale dei due codici è stato plagiato. Andando ad analizzare i metadati relativi alle modifiche dei due documenti, si possono identificare pattern sospetti al di fuori del codice stesso. Come ad esempio la modifica immediata di un documento subito dopo la modifica dell'altro, oppure cambiamenti identici in entrambi i documenti in un lasso di tempo molto breve.
  Utilizzando la libreria GitPython o PyDriller, sarà possibile estrarre i commit relativi ai due documenti per analizzarli in maniera temporale e valutare

di consequenza.
Per far ciò, entrambi i codici sorgenti devono essere sulla piattaforma GitHub (o un qualsiasi altro client git) e che regolarmente gli autori eseguano operazioni di commit e push.

References

1. Python Abstract Syntax Tree docs: `https://docs.python.org/3/library/ast.html`
2. TF-IDF Vectorizer docs: `https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.TfidfVectorizer.html`
3. Cosine similarity docs: `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine\_similarity.html`