Un Approccio Fuzzy per l'Authorship Analysis

Luca Carettoni
luca.carettoni@securenetwork.it

Federico Maggi
f.maggi@securenetwork.it

ABSTRACT

L'analisi del codice sorgente, al fine di caratterizzare il profilo dell'autore di un programma, è una pratica alquanto utilizzata sia in ambito legale che nel settore dell'educazione. Esistono molteplici casi in cui risulta interessante possedere delle metodologie di analisi funzionali. In questo contesto, l'analisi forense è impiegata per determinare lo sviluppatore di un'applicazione o semplicemente per scopi didattici e di valutazione dello stile. Realizzare un profilo dell'autore di un determinato software permette di provare l'appartenenza di parti di codice come prodotto dell'ingegno di tale soggetto. Come è facilmente intuibile, una tale operazione è particolarmente esposta al giudizio soggettivo degli addetti all'analisi. Ultimamente però si sta cercando di automatizzare il processo, definendo delle metriche al fine di poter valutare lo stile di scrittura di un documento. In quest'ottica è possibile definire delle caratteristiche analizzabili tramite logica e misure fuzzy, che riescano a catturare le qualità di forma del codice e dei commenti considerando sia il "modello" di programmazione (uso frequente di variabili temporanee, uso di template, ecc.) che lo stile vero e proprio che ogni autore adotta durante la scrittura di un testo (proporzione delle linee vuote nel testo, lunghezza media degli identificatori, grado di indentazione, ecc.). Inoltre, poichè spesso i dataset risultano piccoli, l'utilizzo di metodi statistici o di reti neurali perde di rilevanza, mentre i sistemi fuzzy sembrano essere uno strumento efficace ed efficiente.

1. INTRODUZIONE

Con il crescente aumento dei casi di infezione da trojan horse, worm e virus a danno dei più diversi sistemi informatici, la tecnica denominata *authorship analysis* e più in generale la *software forensics* [14] ha acquisito un'importanza fondamentale per la lotta ai crimini informatici e telematici. Inoltre, esistono molteplici situazioni in cui si vorrebbe poter identificare l'autore di un frammento di codice; ad esempio, nel caso di dispute legali tra diversi programmatori o nel caso di reingegnerizzazione del codice per verificare se lo stile di programmazione risulta essere quello deciso.

Sebbene il codice sorgente di una qualsiasi applicazione evolva rapidamente nel tempo, subendo integrazioni e modifiche continue, per una parte di programmatori e programmi è possibile cercare di identificarne l'autore con l'impiego delle tecniche sopraccitate. L'individuazione del soggetto è resa possibile dal fatto che le persone, durate la scrittura di software, adottano certi pattern, certe regole di comportamento e di certi stili che rimangono costanti nel tempo. La naturale tendenza dell'uomo è quella di usare ripetutamente i medesimi costrutti e convenzioni, se questi risultano efficaci per il lavoro che sta svolgendo.

Il processo di identificazione è analogo a quello in cui alcune delle caratteristiche di un individuo vengono studiate ed archiviate al fine di identificare la persona successivamente. Esistono, in questo campo, tecniche molto semplici basate sul confronto diretto come ad esempio le fotosegnaletiche, in cui le caratteristiche in gioco sono facilmente modificabili. Esistono tuttavia tecniche decisamente più sofisticate e per questo più costose come il riconoscimento biometrico basato su impronte digitali, la scansione della retina, ecc. Analogamente il processo di identificazione dell'autore di un testo ha l'obiettivo di raccogliere un set di caratteristiche più o meno semplici al fine di aiutare il riconoscimento di un soggetto, in maniera quanto più automatica e relativamente poco costosa.

In letteratura, l'*authorship analysis* per la discriminazione di testi scritti è stata largamente discussa [6,13] ed utilizzata in casi reali per l'attribuzione di paternità di un romanzo ad un autore di cui si conosco le caratteristiche espressive e psicologiche, derivate da opere precedenti di origine certa [15]. Da questi precedenti studi, l'analisi del codice sorgente può trarne solo qualche spunto in quanto il dominio applicativo

è diverso e più intricato: i programmi spesso sono scritti da più persone, sono sviluppati e revisionati in tempi diversi e la loro formattazione è facilmente alterabile mentre la sintassi è rigida.
Il software possiede delle caratteristiche intrinseche [11] che possono essere facilmente quantificate: ad esempio, numero di linee di codice, densità delle istruzioni, lunghezza media dei commenti. D'altra parte, per realizzare un profilo efficace dell'autore di un software, tali caratteristiche non sono sufficienti: è necessario ricavare altre informazioni molto meno *crisp*, difficilmente quantificabili in maniera oggettiva e precisa. Alcune di queste misure riguardano il modo in cui un programmatore approccia un problema, crea un algoritmo, usa ripetutamente variabili temporanee e pattern (pre)definiti. Alcune di queste caratteristiche derivano in primo luogo (1) dalla formazione ricevuta e (2) dall'ambiente in cui si trova ad operare ma anche da alcuni fattori secondari, quali (3) le condizioni psicologiche del soggetto, che favoriscono la modifica dello stile comportamentale rendendo ulteriormente difficile l'analisi.
Da queste considerazioni si può quindi capire come la semplice misurazione automatica spesso non porti a buoni risultati. Dai pochi studi fatti sinora, l'uso di *modelli a regole fuzzy* è stato riconosciuto di grande aiuto nel catturare tutte queste caratteristiche soggettive (le caratteristiche di stile di programmazione dell'autore) che permettono di comporre un profilo veritiero del testo. Spesso poi i dataset disponibili nei casi di *authorship analysis* sono poco numerosi e l'utilizzo di metodi statistici o di reti neurali perde di rilevanza. I modelli fuzzy sembrano invece ancora un metodo efficace sebbene non ancora largamente studiati dalla comunità scientifica.

Scopo di questo lavoro è quello di riassumere gli sforzi di ricerca in questo ambito servendosi anche di un esempio pratico in cui le tecniche presentate vengono impiegate. Nelle Sezioni 2 e 3 si introduce il dominio applicativo, definendo poi in maniera formale, nelle Sezioni 4 e 5, le metriche usate ed il modello che si è cercato di sviluppare.
Per chiarezza è da precisare come, sebbene alcune delle metriche usate sono state ricavate da studi precedenti, la letteratura manchi di esempi completi di un modello fuzzy come quello sviluppato nel corso di questo lavoro. Infine nella Sezione 6 riporteremo dei risultati sperimentali applicando il modello precedentemente proposto. Nella Sezione 7 si cercherà di evidenziare le caratteristiche di questo approccio al problema dell'*authorship analysis* presentando alcuni spunti su possibili sviluppi della ricerca.

2. APPLICAZIONI DELL'ANALISI FORENSE

I principali campi d'uso dell'*authorship analysis*, applicata all'analisi di codice sorgente, possono essere raggruppati in quattro diverse macro categorie [4]: (1) discriminazione dell'autore, (2) identificazione dell'autore, (3) caratterizzazione dell'autore e (4) determinazione d'intenti.

Discriminazione dell'autore

Questa particolare tipologia di analisi è svolta al fine di determinare se uno spezzone di codice sorgente è stato scritto da un singolo o da un gruppo di programmatori. In questo modo è possibile stimare il numero degli autori coinvolti nella scrittura di una singola applicazione; la discriminazione è però anche usata nel caso si voglia mostrare che due pezzi di codice non appartengano allo stesso soggetto, sebbene non identificato.

Identificazione dell'autore

In questo caso l'obiettivo è di verificare, con una certa precisione, se una determinata parte di codice è il frutto dell'ingegno di un individuo ben definito tramite lo studio (a priori) di codice scritto dal medesimo soggetto. Questo tipo di applicazione è molto simile negli intenti alle analisi che vengono svolte su testi letterali al fine di determinare la paternità di un lavoro appena ritrovato. Nel caso della computer forensics è spesso utilizzata per scoprire l'identità di un soggetto che ha scritto un nuovo pezzo di codice, spesso un virus, analizzando repository di codice sorgente scritto da diversi soggetti indiziati.

Caratterizzazione dell'autore

Nel caso della caratterizzazione, si vuole determinare una serie di caratteristiche associabili al programmatore che ha scritto un particolare frammento di codice; realizzare un profilo del programmatore significa poter comprendere in parte la personalità, la formazione ricevuta e l'esperienza acquisita tramite lo studio dello stile di programmazione.

Determinazione d'intenti

In alcuni casi è possibile definire attraverso l'analisi del codice se un "effetto indesiderato" del software è da considerare come accidentale o volutamente realizzato dal programmatore. Poichè il processo di sviluppo di software è spesso rallentato dalla presenza di bug, i cui effetti hanno spesso conseguenze catastrofiche in termini economici, questo tipo di domande sono frequenti. Determinare la negligenza di un soggetto o la sua colpevolezza è il primo passo per poter compiere delle azioni legali.

Per completezza vogliamo sottolineare che nelle tecniche forensi, tutte le analisi sopra riportate, non si limitano all'uso di codice sorgente ma possono trarre informazione anche dall'eventuale decompilazione parziale di eseguibili o dalla traduzione di un linguaggio di più basso livello, come nel caso dei linguaggi interpretati. Per i nostri scopi, l'analisi sarà limitata solamente alla pura analisi del codice sorgente.

3. MOTIVAZIONI

Riconosciuta la possibilità di fare *authorship analysis* nel caso sussistano alcune condizioni essenziali (presenza del codice sorgente, frammenti di codice sufficientemente estesi, ecc.) giustificheremo ora la sua applicabilità pratica mostrando i benefici che l'uso di questa tecnica può portare.

Ambito legale

Nelle azioni legali, c'è la necessità di usare metodologie che possano fornire empiricamente degli indizi per risolvere dispute legali. Con l'aumento dei casi di crimini informatici è sempre più importante avere a disposizione delle tecniche che possano fornire nuove informazioni per determinare l'autore di un virus, ad esempio. Poichè ci si aspetta che in futuro la frequenza di questi crimini continuerà ad aumentare, risulta indispensabile studiare e realizzare tecniche robuste ed automatiche per far fronte a questo fenomeno.

Ambiente accademico

Negli ambienti accademici esiste spesso la necessità di determinare se due elaborati sono "uguali" [5, 16]. Per questo particolare dominio applicativo le tecniche di *plagiarism detection* sarebbero in grado di individuare le differenze tra due programmi, anche in presenza di modifiche stilistiche (modifica dei commenti, diversa indentazione, ridenominazione delle variabili, sostituzione di costrutti con altri equivalenti). Le tecniche di *authorship analysis* possono fare molto di più, in quanto è possibile determinare se i due testi sono stati scritti da persone diverse ed eventualmente individuare la persona che è realmente l'autore. Ad esempio, i due frammenti seguenti possono essere considerati simili secondo le tecniche di *plagiarism detection* ma diversi secondo l'applicazione di *authorship analysis*.

```
/*****************************************
 * Subroutine for checking a string for TABs
 * Parameters: s1:String to examine
 * Return: Boolean indicating the existence
 *       of a tab character
 *****************************************/
 int strchk(char *s1)
 {
  char *ptr1;

  ptr1 = s1;
  while(*ptr1 != 0)
    if(*ptr1++ == '\t')
      return(1);
  return(0);
 }
```

```
#define TRUE 1
#define FALSE 0
/* Checks the existence of \t in string */
int check_for_tab_in_string(string)
{
  char *char_pointer;

  /* Loop until found or we reach EOLN */
  for(char_pointer = string;*char_pointer != NULL;)
  {
    /* check to see if we found TAB */
    if(*(cha_pointer++) == 9)
    {
      /* Success!! */
      return(TRUE);
    }
  }
  /* No tab */
  return(FALSE);
}
```

A fianco dei problemi di plagio esiste, negli ambiente accademici legati all'information technology, l'esigenza di verificare lo stile di programmazione degli allievi. Queste tecniche possono anche verificare la somiglianza di un frammento di codice a quella di un campione di codici sorgente "buoni".

Industria del software

Nell'industria del software, lo sviluppo di alcune applicazioni dura da anni lungo una serie continua di modifiche e reintegrazioni di codice scritto da milioni di persone. Quando un particolare modulo software o un programma deve essere riscritto è necessario determinare l'autore di tale parte; determinare tale soggetto, tra gli innumerevoli programmatori che hanno preso parte allo sviluppo del software, è essenziale per un buon processo di reingegnerizzazione in quanto nessuno meglio dell'autore stesso conosce l'architettura di tale modulo.

Rilevazione real-time di anomalie

Nel caso di sistemi real-time per la *misuse detection*, abbinare tecniche di *authorship analysis* alle altre già consolidate per questo tipo di attività, può aumentare l'efficacia di tali strumenti. Realizzare un profilo d'uso in condizioni normali, tramite l'analisi delle caratteristiche del programma, costituisce un pattern che può essere usato per monitorare la presenza di usi anomali [8].

4. METRICHE DEL SOFTWARE

Sebbene i linguaggi di programmazione ammettano una minor libertà rispetto ai linguaggi naturali, in termini di lessico e sintassi, hanno comunque una flessibilità espressiva tale da permettere ai programmatori di personalizzare il codice. Per fornire qualche esempio potremmo pensare ad un programmatore che preferisce un particolare costrutto di *loop* mentre altri normalmente ne utilizzano alcuni, sintatticamente diversi, ma (funzionalmente) equivalenti al primo; oppure un programmatore che curi meticolosamente l'indentazione del suo codice, rispetto ai suoi colleghi.

Di seguito sono mostrati due frammenti di codice [4] scritti in C++ da due diversi programmatori. Entrambi i programmi calcolano la funzione matematica fattoriale di un numero n.

```
// Factorial takes an integer as an input
// and returns the facctorial of the input.
// This routine does not deal with negative values!

int Factorial (int Input)
{
```

```
    int Counter;
    int Fact;
    Fact=1; //Initialises Fact to 1: factorial 0 is 1
    for (Counter=Input; Counter>1; Counter=Counter-1)
    {
        Fact=Fact*Counter;
    }
    return Fact;
}
```

```
int f(int x){
int a, y=1;
if (!x) return 1; else return x*f(x-1);}
```

Lo stesso problema è stato risolto dai due programmatori in maniera diversa: sia con diversi algoritmi che diverso stile di programmazione. Le differenze stilistiche includono (1) l'uso dei commenti, (2) i nomi delle variabili, (3) l'uso di spazi bianchi, (4) l'indentazione e (5) il livello di leggibilità di ogni funzione.
Questi frammenti, sebbene estremamente corti, illustrano come sia possibile identificare delle caratteristiche proprie del codice che possono aiutare nella discriminazione tra due autori.

Prima di definire le metriche che useremo nel nostro modello a regole fuzzy è opportuno identificare le grandezze caratteristiche del codice sorgente di un'applicazione, in maniera da valutare quelle oggettivamente misurabili rispetto a quelle che necessitano di una rappresentazione più sfumata (soggettiva), che quindi ben si prestano per un'analisi fuzzy. È inoltre necessario valutare quali di queste metriche siano effettivamente utilizzabili per l'*authorship analysis* ovvero quali rappresentino delle grandezze discriminatorie tra lo stile di un autore ed un altro. In questa Sezione forniremo solamente un accenno a tali proprietà; il lettore interessato può fare riferimento ad alcuni testi specifici sull'argomento [1, 14].

Krusl e Spafford [9] hanno mostrato come alcune caratteristiche tipografiche (o stilistiche) del codice siano poco interessanti per la caratterizzazione di un testo, in quanto poco variabili tra un programmatore ed un altro. Dopo aver analizzato 1000 file scritti in linguaggio C, da brevi spezzoni di poche linee di codice sino a programmi di oltre 7000 linee, hanno calcolato tramite un programma automatico il numero di linee bianche, costruendo la distribuzione statistica riportata in Figura 1. Da questo grafico è possibile notare come la distribuzione percentuale sia maggiormente addensata intorno al 10% e solo pochi campioni presentino una proporzione diversa di linee bianche rispetto alle linee di codice formate da istruzioni. Da esperimenti come questi si verifica quindi che tale metrica è alquanto debole poichè molto simile tra diversi campioni di applicazioni, scritte da diversi programmatori.

Da un simile esperimento si evince invece come l'indentazione del codice possa essere considerata, in linea di massima,

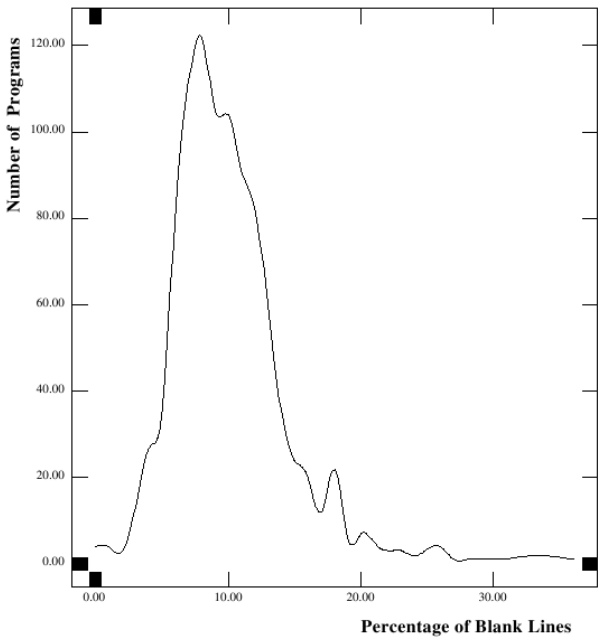

Figura 1: Distribuzione delle linee bianche, all'interno di programmi scritti in C

un buon parametro caratterizzante. Analizzando altri 178 programmmi C, da poche sino a 5000 linee di codice, gli autori costruiscono la distribuzione del valor medio e massimo d'indentazione. Dai grafici riportati in Figura 2 e 3 si nota l'enorme variabilità della metrica sebbene la maggior parte dei programmatori usi due, quattro o otto spazi (*Tab Space*) di indentazione.

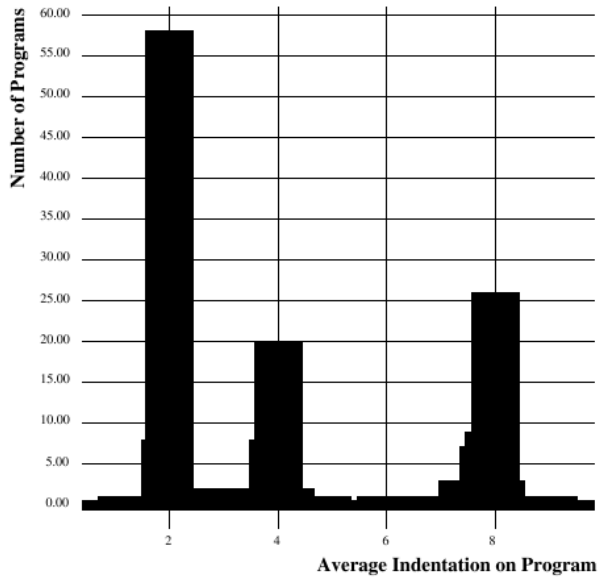

Figura 2: Indentazione media, all'interno di programmi scritti in C

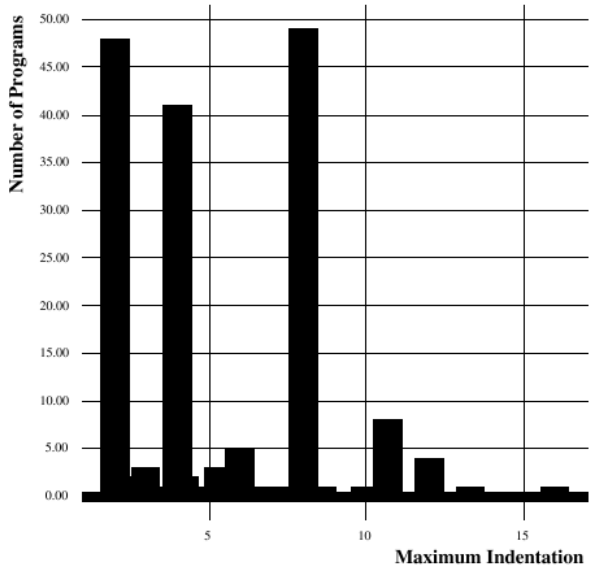

Figura 3: Indentazione massima, all'interno di programmi scritti in C

Per questa caratteristica è però opportuno ricordare che molti degli *Integrated Development Environment* (IDE), utilizzati dagli sviluppatori oggigiorno, forniscono delle funzionalità di indentazione automatica che possono quindi modificare la normale indentazione che l'autore del software userebbe durante la stesura dello stesso.

Un altro parametro spesso utilizzato in studi di questo tipo è l'uso di *naming convention*. Molti programmatori utilizzano un approccio classico durante l'assegnamento di nomi alle variabili, che è fortemente dipendente dal linguaggio di programmazione considerato (es: `myPrivateVariable`, `temp1` per il linguaggio Java [10]). Altri usano un identificativo fisso più una parte variabile, magari incrementale, per determinare l'uso o la visibilità della variabile (es: `1_var`, `2_var` oppure `size_module` e `range_module`). Altri ancora invece non hanno uno stile fisso. Anche nell'ultimo caso, la completa assenza di una convenzione è comunque una caratteristica tipografica importante che determina un preciso stile di scrittura del codice e determina una sorta di "brand" personale dell'autore.

La lunghezza del programma, in termini di numero di linee di codice, è una caratteristica che risulta utile solamente nel caso in cui, durante l'analisi forense, si abbia a disposizione un programma simile (in termini di requisiti funzionali) scritto dallo stesso autore. Sebbene possa sembrare una situazione poco realistica è da considerare come sia comune, per gli sviluppatori professionisti che da anni svolgono tale professione, costruire e testare delle librerie di funzioni comunemente usate in gran parte delle applicazioni (ad esempio, i moduli che effettuano il login, le librerie che si occupano di creare e gestire i log); se il contesto applicativo è lo stesso, il riuso di tali spezzoni di codici adattati per l'esigenza è una pratica comunemente adottata per ridurre i tempi di sviluppo.

Prima di concludere questa Sezione, ci sembra opportuno riportare un elenco ordinato delle *caratteristiche* proprie del codice sorgente dalle quali ricaveremo le metriche da utilizzare nel nostro modello fuzzy. Alcuni di questi parametri, specificati brevemente di seguito, sono stati selezionati da studi precedenti presenti in letteratura [7, 9].

C.1 Numero di linee che sono, o che includono, commenti rispetto al numero totale delle linee di codice. Un semplice conteggio del numero di linee di commento.

C.2 Numero di linee di commento che sono sulla stessa linea di istruzioni rispetto al numero totale di linee di commento.

C.3 Numero di linee di commento che sono contornati da un bordo, formato da uno o più caratteri ripetuti, rispetto al numero totale di linee di commento.

C.4 Numero di identificatori che sono descritti da commenti successivi o precedenti, rispetto al numero totale di identificatori. Per la valutazione di questa metrica, i commenti devono essere ovviamente inerenti all'identificatore stesso.

C.5 Lunghezza media degli identifcatori rispetto alla lunghezza dell'identificatore più lungo.

C.6 Uso di template o costrutti di programmazione ricorrenti.

C.7 Numero di istruzioni medie per funzioni/metodi rispetto al massimo numero di istruzioni in un singolo metodo o in una singola funzione. Esula da questo conteggio il numero di istruzioni presenti nel `main`.

C.8 Numero medio di istruzioni multiple per linea rispetto al numero totale di linee del codice sorgente. Questa caratteristica può essere ricavata semplicemente con un conteggio sul numero di caratteri di *fine istruzione* sulla medesima linea.

C.9 Numero di identificatori con il carattere *underscore* rispetto al numero totale di identificatori.

C.10 Numero di identificatori con il primo carattere del nome appartenente ad [A-Z] rispetto al numero totale di identificatori.

C.11 Numero di caratteri in minuscolo rispetto al numero totale di caratteri. In questo conteggio si

considera il codice sorgente nella sua completezza (commenti compresi).

C.12 Numero di linee bianche rispetto al numero totale delle linee di codice. Sebbene, come precedentemente affermato, questa caratteristica è significativa solo in un numero ridotto di casi verrà riportata in questo elenco per completezza.

C.13 Numero di istruzioni condizionali indentante su linee diverse rispetto al numero totale di linee. In questo conteggio sono considerate tutte quelle istruzioni condizionali che possono essere costituite da un numero di rami maggiore di uno.

C.14 Grado di indentazione usata.

C.15 Significatività del nome assegnato agli identificatori.

C.16 Uso di variabili temporanee.

C.17 Errori di ortografia o di forma presenti nei commenti.

C.18 Qualità dei commenti e sua attinenza rispetto al codice inerente.

5. METRICHE FUZZY

I maggiori vantaggi nell'usare variabili fuzzy, per descrivere modelli, derivano dalla loro naturale capacità di catturare semanticamente concetti soggettivi attraverso delle semplici definizioni qualitative. Molte delle caratteristiche del software sono misurabili quantitativamente tramite strumenti automatici mentre altre sono caratteristiche complesse e difficilmente esprimibili attraverso una rappresentazione binaria (esiste o meno tale proprietà) o tramite un numero. Un altro vantaggio, usando variabili fuzzy, è la possibile riduzione del numero di parametri liberi del modello da dover calibrare; in questo modo è possibile tarare il modello con pochi campioni e l'analisi è quindi attuabile anche su dataset ridotti.

La presenza di alcune caratteristiche del software che sono difficilmente quantificabili oggettivamente, ma che possono essere interpretate in maniera naturale tramite metriche fuzzy, ci suggerisce di creare un modello combinando i tradizionali metodi di misura e il nuovo approccio basato appunto sulla logica fuzzy. Volendo generalizzare si potrebbe includere tutte le metriche in un'unica categoria: le metriche tradizionali (oggettive) avranno grado di appartenenza pari ad 1 (caso degenere di insieme fuzzy). I programmatori esperti possono facilmente analizzare le applicazioni e abbinare delle valutazioni (soggettive), con un certo grado di attinenza, su talune proprietà. Valutando nuovamente le metriche proposte in letteratura e presentate nella Sezione 4 possiamo identificare quelle variabili (caratteristiche) che necessitano di una rappresentazione qualitativa e che faranno parte del nostro modello (parte fuzzy). Successivamente

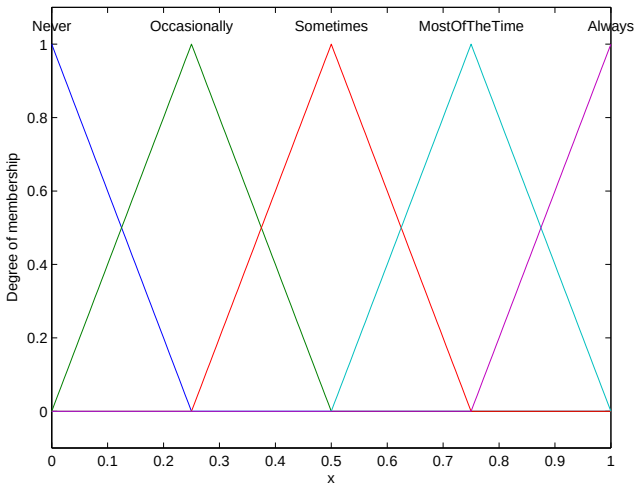

Figura 4: Insiemi fuzzy (funzioni di appartenenza) comuni alle variabili C.6, C.15—C.18

ogni metrica/variabile verrà dettagliata con più precisione anche grazie alla relativa funzione di appartenenza (in Figura 4) che è uguale per tutte le variabili. Per le altre variabili (caratteristiche) è implicita una funzione di appartenenza crisp.

C.6 Uso di template o costrutti di programmazione ricorrenti.

C.14 Grado di indentazione usata.

C.15 Significatività del nome assegnato agli identificatori.

C.16 Uso di variabili temporanee.

C.17 Errori di ortografia o di forma presenti nei commenti.

C.18 Qualità dei commenti e sua attinenza al codice inerente.

Per tali variabili sono definite le seguenti etichette:

Valore fuzzy	*Abbrev.*
Never/Almost Never	N
Occasionally	O
Sometimes	S
Most of the time	M
Always/Almost Always	A

Tabella 1: Fuzzy Value definiti per le variabili caratteristiche

La metrica C.6 indica dei blocchi di codice riconducibili ad alcuni design pattern. Il grado di aderenza al pattern è

una caratteristica difficilmente misurabile ma che esprimiamo con *Never/Almost Never* se in quasi nessun caso c'è una stretta aderenza a qualche modello oppure all'estremo opposto con *Always/Almost Always* se quasi tutta l'applicazione analizzata può essere riconducibile ad un template. Con questa caratteristica si vuole evidenziare il grado di riuso di soluzioni algoritmiche consolidate tra programmatori professionisti: blocchi di codice riconducibili ad alcuni design pattern classici nell'ingegneria del software [3] o semplici strutture o costrutti utilizzati più volte nello stesso frammento di codice analizzato.

La metrica C.15 è molto semplice ed indica la profondità di annidamento. La metrica C.16 invece è difficilmente misurabile —senza intervento umano: cerca infatti di esprimere il grado di significatività/leggibilità dei nomi scelti per gli identificatori. Ad esempio, le variabili `ptr1`, `char_pointer`, `char_pointer_1`, pur indicando lo stesso concetto (i.e. il primo puntatore a carattere dichiarato), hanno grado di significatività crescente; se nel corso di un frammento di codice compaiono variabili nella forma di `ptr1` allora la metrica C.16 ha etichetta "N". Similmente per gli altri due casi.

Per quanto riguarda la metrica C.17, sono considerati come errori sia i commenti grammaticalmente scorretti che quelli che risultano corretti ma poco chiari a livello di comprensione. Pertanto, un correttore grammaticale (ingl. *spell checker*) è solo il primo passo per la misurazione di questa metrica; un intervento umano è quasi sempre necessario.

La misurazione automatica della metrica C.18 è pressoché impossibile: solo un intervento umano è in grado di stabilire (anche con bassa precisione) l'effettiva attinenza di un commento al codice di appartenenza, sia per ragioni semantiche che per ragioni linguistiche.

6. CASO DI STUDIO

Come anticipato, nella letteratura presa in considerazione non esiste un caso di studio esaustivo e completo; il più ricco di risultati è quello presentato in [7] che —tuttavia— non va oltre la raccolta dei dati. Probabilmente, il motivo principale di questa mancanza sono le non poche problematiche che si debbono risolvere per riuscire ad ottenere dei risultati. Ad esempio, per quanto riguarda il rilevamento di plagio, è difficile avere a disposizione *altro* codice di confronto relativo ai due autori. In generale, non è detto che si disponga di altri risultati con cui confrontare.

Altra e più importante questione da risolvere è, a nostro avviso, quella relativa a come "aggregare" i due contributi: (1) le analisi fatte sulle metriche non-fuzzy e (2) quelle sulle metriche fuzzy. Come anticipato, si potrebbe interpretare le variabili non-fuzzy alla stregua di variabili fuzzy degeneri (con funzione di appartenenza costante e pari a 1). Questa soluzione presenta uno svantaggio principale. In particolare, relativamente al confronto tra due codici sorgenti (al fine di asserirne l'appartenenza, o meno, allo stesso autore), è altamente improbabile che due misure reali non-fuzzy abbiano esattamente lo stesso valore, seppur "vicine". Ad esempio, è universalmente accettato che, con una precisione del decimo, i valori 0.3 e 0.31 sono identici; sono invece diversi se la precisione aumenta. Similmente, l'importanza delle varie metriche (non-)fuzzy può variare a seconda delle applicazioni, del tipo di linguaggio di programmazione e di altre eventuali particolari situazioni in cui viene svolto il test.

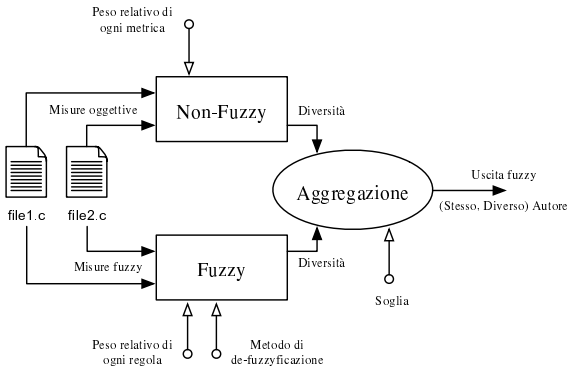

Figura 5: Modello di riferimento per l'analisi combinata di metriche fuzzy e non-fuzzy

In generale, dalle considerazioni fatte si nota la necessità di realizzare un modello "sensato" secondo dei parametri non oggettivi, fortemente dipendenti dal contesto applicativo e che possono, eventualmente, essere appresi tramite una rete neurale. Pertanto, per quanto riguarda le metriche da impiegare e sulle modalità di misurazione ci siamo affidati ai risultati presenti in letteratura [7, 9], spostando invece l'attenzione sulla realizzazione di un modello con le caratteristiche sopraccitate; modello, presentato in Figura 5, che verrà descritto in seguito nelle sue parti ed impiegato in un caso reale. Si rammenta che la maggior parte delle misure sono state effettuate manualmente, anche se alcune (ma non tutte), sono automatizzabili come accennato in precedenza.

Per quanto riguarda i dati in ingresso (file sorgente) sono stati impiegati tre (frammenti di) programmi scritti —in ambito accademico— da due diversi autori provenienti dallo stesso corso di studi. Le ipotesi sono quelle di due autori con una formazione di base comune, con abilità medio/alte nell'attività di programmazione in piccolo. Il fatto che i frammenti siano scritti in linguaggio C ci ha permesso di raccogliere quante più metriche possibili, essendo il C un linguaggio piuttosto classico, "generico" e con vaste possibilità applicative.

Parte non-fuzzy

Questo blocco si occupa dell'analisi delle metriche non-fuzzy, attraverso l'impiego di criteri semplici e —più che altro— basati sul "buon senso" (e che comunque trovano giustificazione nella teoria della stima e della verifica di ipotesi

statistiche [12]). In breve, ricevendo in ingresso le misure delle metriche non-fuzzy dei due file sorgente, deve restituire una misura di "similarità/diversità" delle due tuple di valori. Non avendo a disposizione altre informazioni non è possibile effettuare dei test particolarmente intelligenti. Come anticipato, sarebbe necessario effettuare delle analisi statistiche su vasti campioni per poter individuare dei valori "normali" di riferimento e fornire delle misure di paragone. Nell'ipotesi di scarsa conoscenza iniziale sui dati da confrontare ci si può accontentare di esprimere la similarità/diversità come la media pesata delle norme tra ogni coppia di misure (normalizzate). Ovviamente, modificando opportunamente i pesi, è sempre possibile "violare" il significato di questa misura, nel senso che basterebbe far pesare poco le metriche che influiscono negativamente sul risultato e viceversa. Tuttavia, l'intento è quello di dare la possibilità di *tarare* il sistema in modo che —nel caso sia possibile effettuare delle statistiche— i pesi siano "sensati" rispetto —per esempio— al tipo di linguaggio di programmazione utilizzato nel frammento. Ad esempio, si può spezzare un frammento di codice (molto lungo) in più parti per calcolare su di esse delle statistiche, simulando quindi la presenza di più campioni anzichè uno. Nella Sezione 7 è stata comunque studiata l'applicabilità di soluzioni alternative.
Se $\mathbf{x}$ e $\mathbf{y}$ sono le due tuple e $\mathbf{w}$ è la tupla dei pesi di ogni metrica, allora in uscita dal blocco si avrà

$$z = 1 - \frac{\sum_j w_j |x_j - y_j|}{\sum_j w_j} \tag{1}$$

Parte fuzzy

Il confronto delle metriche non oggettive si basa su un sistema a regole fuzzy i cui ingressi sono, appunto, le misure delle metriche e l'uscita, de-fuzzyficata, del grado di similarità dei due set di misure. I fuzzy-set relativi alle metriche (variabili) in ingresso sono stati già definiti nella Sezione 5 e sono tutti identici. Rimangono da definire (1) il tipo di regole da impiegare e (2) il meccanismo inferenziale secondo cui vengono valutate. Per rimanere aderenti alle scelte adottate in precedenza, si vuole lasciare la possibilità di personalizzare il sistema; in questo caso si può intervenire (1) sul peso relativo di ogni regola rispetto alle altre e (2) sul metodo di de-fuzzyficazione (es. centroide) che —in molti casi— può influenzare di molto l'uscita. Le altre caratteristiche del sistema sono quelle tipiche per applicazioni generiche, ossia: sistema a regole di tipo *mamdani* e T-norme *minimo, massimo, minimo, massimo* per la valutazione di AND, OR, $\Rightarrow$, aggregazione, rispettivamente.

Dal momento che l'interesse è quello di valutare la "similarità/lontananza" tra due metriche, le regole saranno del tipo

1. If (x is Never) and (y is Never) then (z is Similar)
2. If (x is Occasionally) and (y is Occasionally) then (z is Similar)
3. If (x is Sometimes) and (y is Sometimes) then (z is Similar)
4. If (x is MostOfTheTime) and (y is MostOfTheTime) then (z is Similar)
5. If (x is Always) and (y is Always) then (z is Similar)
6. If (x is Never) and (y is not Never) then (z is Different)
7. If (x is Occasionally) and (y is not Occasionally) then (z is Different)
8. If (x is Sometimes) and (y is not Sometimes) then (z is Different)
9. If (x is MostOfTheTime) and (y is not MostOfTheTime) then (z is Different)
10. If (x is Always) and (y is not Always) then (z is Different)
11. If (x is Always) and (y is not Always) then (z is Different)

dove x, y è la coppia di misure della stessa metrica da confrontare, il cui grado di similarità è espresso dall'uscita fuzzy z. La forma del fuzzy-set della variabile z è rappresentata in Figura 6. Per ogni metrica esiste quindi un sistema di regole fuzzy il cui compito è quello di confrontare le due misure provenienti dai due file sorgente. Si noti che gli ingressi di questo blocco non sono i valori (normalizzati) di ciascuna delle variabili, ma direttamente la coppia "fuzzy-set"—"grado di appartenenza", dal momento che lo scopo vero e proprio di questo approccio è dare la possibilità di esprimere dei valori soggettivi per ogni metrica.

Per fornire un'unica uscita è necessario aggregare, sempre tramite una media pesata, il valore de-fuzzyficato, di ogni uscita.

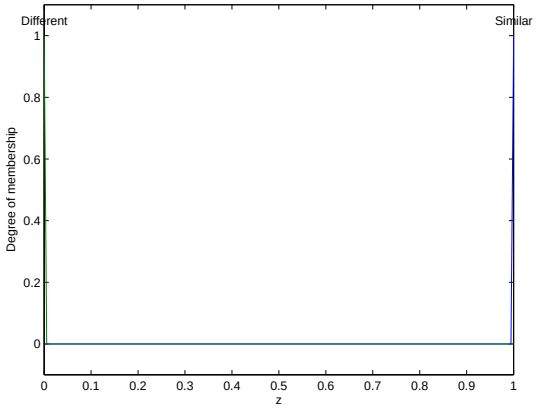

Figura 6: Fuzzy-set di uscita, per esprimere "similarità/dissimilarità" tra due metriche confrontate in ingresso

Il blocco di aggregazione, rappresentato da un ellisse, si occupa di fornire una misura aggregata (media aritmetica) dei risultati provenienti dai due blocchi appena descritti.

Dati raccolti e risultati

Riportiamo di seguito i confronti eseguiti tra tre coppie di file. I passi compiuti possono essere così riassunti:

1. Misurazione manuale e automatica delle m metriche non-fuzzy, per entrambi i file sorgente. Il risultato è una coppia di tuple: $\mathbf{x} = \langle x_1, x_2, ..., x_m \rangle$ e $\mathbf{y} = \langle y_1, y_2, ..., y_m \rangle$.
2. Misurazione manuale delle n metriche fuzzy, per entrambi i file sorgente. Il risultato è una coppia di tuple $\tilde{\mathbf{x}} = \langle \tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n \rangle$ e $\tilde{\mathbf{y}} = \langle \tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_n \rangle$. Dove, ad esempio, $\tilde{x} = (N, 0.3 + O, 0.7)$.
3. Confronto delle coppie di (misure delle) metriche fuzzy.
   (a) Confronto di ogni coppia relativamente alla stessa metrica.
   (b) Calcolo della media pesata dei risultati di tutti i confronti. Il risultato, dopo la de-fuzzyficazione, è $S \in [0, 1]$ (grado di similarità).
4. Confronto delle coppie di (misure delle) metriche *non* fuzzy.
   (a) Confronto di ogni coppia relativamente alla stessa metrica.
   (b) Calcolo della media pesata dei risultati di tutti i confronti. Il risultato è $s \in [0, 1]$ (similarità).
5. Calcolo dell'uscita in $[0, 1]$ alla quale, eventualmente, è possibile (1) dare un'interpretazione tramite fuzzy-set oppure (2) mapparla in $\{VERO, FALSO\}$ specificando una soglia. Ad esempio, se l'uscita è ≥ 0.6 allora $VERO$ (stesso autore).

Eventualmente, può seguire una fase di apprendimento supervisionato dei parametri del sistema.

Le misure relative ai tre file sorgente sono raccolte nella Tabella 2. I confronti che sono stati effettuati sono tre: (1) `file1.c` versus `file2.c` (stesso autore), (2) `file1.c` versus `file3.c` (diverso autore) e (3) `file2.c` versus `file3.c` (diverso autore).

Relativamente ai tre esperimenti, gli ingressi del sistema in Figura 5 sono rispettivamente i seguenti:

Esperimento 1 (`file1.c` versus `file2.c`, stesso autore)

$$\mathbf{x} = \langle 21/100, 0/21, 0/21, 0/16, 9.625/15, (N, 0.2 + O, 0.8), 50/50, 0/100, 0/16, 14/16, 294/1658, 25/100, 4/4, (M, 0.3 + A, 0.7), (S, 0.2 + M, 0.8), (N, 0.4 + O, 0.6), (N, 0.8 + O, 0.2), (O, 0.3 + S, 0.7) \rangle$$

	`file1.c`	`file2.c`	`file3.c`
C.1	21/100	17/120	12/101
C.2	0/21	0/17	6/12
C.3	0/21	0/17	0/12
C.4	0/16	0/3	1/12
C.5	9.625/15	5/5	1.83/6
C.6	O, 0.8 + N, 0.2	A, 0.8 + M, 0.2	N, 1
C.7	50/50	26.3/79	29.5/59
C.8	0/100	0/120	2/101
C.9	0/16	0/3	0/12
C.10	14/16	3/3	0/12
C.11	294/1658	537/2169	34/1094
C.12	25/100	22/120	10/101
C.13	4/4	12/12	6/6
C.14	A, 0.7 + M, 0.3	A 0.9 + M, 0.1	S, 0.8 + 0, 0.2
C.15	M, 0.8 + S, 0.2	A, 0.7 + M, 0.3	O, 0.6 + N, 0.4
C.16	O, 0.6 + N, 0.4	N, 0.9 + O, 0.1	O, 0.9 + S, 0.1
C.17	N, 0.8 + O, 0.2	N, 0.9 + O, 0.1	O, 0.8 + S, 0.2
C.18	S, 0.7 + O, 0.3	N, 0.9 + O, 0.1	S, 0.9 + O, 0.1

Tabella 2: Misurazione su tre file sorgente, scritti da due autori differenti.

$$\mathbf{y} = \langle 17/120, 0/17, 0/17, 0/3, 5/5, (M, 0.2 + A, 0.8), 26.3/79, 0/120, 0/3, 3/3, 537/2169, 22/120, 12/12, (M, 0.1 + A, 0.9), (M, 0.3 + A, 0.7), (N, 0.9 + O, 0.1), (N, 0.9 + O, 0.1), (N, 0.9 + O, 0.1) \rangle$$

Esperimento 2 (`file1.c` versus `file3.c`, diverso autore)

$$\mathbf{x} = \langle 21/100, 0/21, 0/21, 0/16, 9.625/15, (N, 0.2 + O, 0.8), 50/50, 0/100, 0/16, 14/16, 294/1658, 25/100, 4/4, (M, 0.3 + A, 0.7), (S, 0.2 + M, 0.8), (N, 0.4 + O, 0.6), (N, 0.8 + O, 0.2), (O, 0.3 + S, 0.7) \rangle$$

$$\mathbf{y} = \langle 12/101, 6/12, 0/12, 1/12, 1.83/6, (N, 1 + O, 0), 29.5/59, 2/101, 0/12, 0/12, 34/1094, 10/101, 6/6, (O, 0.2 + S, 0.8), (N, 0.4 + O, 0.6), (O, 0.9 + S, 0.1), (O, 0.8 + S, 0.2), (O, 0.1 + S, 0.9) \rangle$$

Esperimento 3 (`file2.c` versus `file3.c`, diverso au-

tore)

$$\mathbf{y} = \langle 17/120, 0/17, 0/17, 0/3, 5/5, (M, 0.2 + A, 0.8), 26.3/79, 0/120, 0/3, 3/3, 537/2169, 22/120, 12/12, (M, 0.1 + A, 0.9), (M, 0.3+A, 0.7), (N, 0.9+O, 0.1), (N, 0.9+O, 0.1), (N, 0.9 + O, 0.1)\rangle$$

$$\mathbf{y} = \langle 12/101, 6/12, 0/12, 1/12, 1.83/6, (N, 1 + O, 0), 29.5/59, 2/101, 0/12, 0/12, 34/1094, 10/101, 6/6, (O, 0.2 + S, 0.8), (N, 0.4+O, 0.6), (O, 0.9+S, 0.1), (O, 0.8+S, 0.2), (O, 0.1 + S, 0.9)\rangle$$

Risultati

La Tabella 3 riporta l'output di ogni esperimento, corredato con gli output dei blocchi intermedi.

Grado di diff. non-fuzzy:	0.070202
Grado di diff. fuzzy:	0.576562
Esito fuzzy:	Stesso autore-0.676618
Esito atteso:	Stesso autore

Grado di diff. non-fuzzy:	0.213479
Grado di diff. fuzzy:	0.756250
Esito fuzzy:	Stesso autore-0.515136
Esito atteso:	Diverso autore

Grado di diff. non-fuzzy:	0.254281
Grado di diff. fuzzy:	0.954861
Esito fuzzy:	Diverso autore-0.604571
Esito atteso:	Diverso autore

Tabella 3: Risultati degli esperimenti

Sebbene il numero di esperimenti che è stato possibile condurre sia molto limitato, dai risultati si nota immediatamente come l'analisi delle metriche non-fuzzy riconosca meglio la *non-diversità* mentre l'analisi tramite regole fuzzy riconosca meglio la *diversità* degli autori. Nel primo esperimento infatti, l'analisi delle metriche non-fuzzy rileva un basso grado di differenza tra gli autori mentre il sistema di regole fuzzy *sbaglia* riportando un grado di differenza prossimo al 60%.

Viceversa, nel secondo esperimento la parte fuzzy rileva, correttamente, un grado di diversità alto (maggiore del 75%) mentre le metriche non-fuzzy sbagliano rilevando un basso grado di diversità, tale da portare un risultato finale errato. Tale divergenza è ancora più marcata nel terzo esperimento, in particolare per quanto riguarda la parte fuzzy la quale rileva un grado di diversità degli autori prossimo al 100%.

7. CONCLUSIONI

Nel corso di questo lavoro ci siamo preposti di studiare l'authorship analysis con particolare attenzione all'impiego di logica/metriche fuzzy per il confronto di caratteristiche non oggettive.

Dopo aver scelto e definito le metriche da utilizzare, l'obiettivo è stato quello di realizzare un modello completo che permettesse di effettuare delle prove, con lo scopo di valutare la bontà dell'approccio. Come già sottolineato, l'uso di logiche fuzzy in questo contesto è di recente introduzione in letteratura, pertanto il modello realizzato vuole essere soltanto un primo tentativo verso la definizione di un processo per l'analisi aggregata delle diverse metriche.

Per il motivo appena presentato non è possibile asserire con sicurezza la validità dei risultati sperimentali. Inoltre, la valutazione delle metriche per casi di test reali non è facilmente effettuabile manualmente in tempi brevi e quindi le prove sono state svolte su un dataset ridotto (i.e. ridotto numero di linee di codice, scarsa varietà di autori). Essendo l'analisi tramite logica fuzzy applicabile su dataset ridotti, tali problemi hanno influito particolarmente durante l'analisi delle metriche non-fuzzy.

Uno dei principali problemi incontrati è stato pertanto l'impossibilità di impiegare delle distanze —tra metriche oggettive— meno banali della classica distanza euclidea, la quale perde facilmente di significato senza valori di riferimento —precedentemente acquisiti— relativi allo stesso autore (o con profilo simile).

Si consideri che tuttavia, nelle applicazioni reali di authorship analysis il contesto è il medesimo; ossia non si dispone di conoscenza pregressa relativa ad un singolo autore. Nel migliore dei casi si ha a disposizione un frammento di codice molto esteso; tale frammento può essere opportunamente frazionato emulando delle realizzazioni pseudo-indipendenti relative allo stesso programmatore dalle quali è possibile calcolare delle statistiche utilizzabili come valori di riferimento.

Una possibile alternativa, la cui trattazione esula dallo scopo di questo lavoro, consiste nell'impiego di tecniche di MultiDimensional Scaling (MDS) [2]. Implementate in alcuni pacchetti statistici, tali tecniche aiuterebbero gli analisti a rivelare il significato intrinseco delle dimensioni secondo cui le dissimilarità tra oggetti (misure delle metriche) sono valutate.

I risultati sperimentali ottenuti, sebbene affetti da imprecisioni e semplificazioni sui dati, hanno mostrato come sia realmente possibile estrarre delle informazioni nascoste dall'analisi di spezzoni di codice sorgente. I linguaggi di programmazione definiscono in maniera formale la sintassi e la semantica dei programmi, ma presentano comunque un cer-

to grado di libertà espressiva che risulta sufficiente per poter estrarre informazioni relative all'autore del testo.

I test sperimentali hanno inoltre evidenziato come i risultati siano fortemente dipendenti dai pesi assegnati a ciascuna metrica, suggerendo l'uso di tecniche di apprendimento per la calibrazione. Possibili sviluppi futuri dovranno quindi studiare degli opportuni meccanismi di taratura, per migliorare la qualità dell'approccio e confermare la validità del metodo con test estensivi.

8. BIBLIOGRAFIA

[1] R. E. Berry and B. A. Meekings. A style analysis of c programs. *Commun. ACM*, 28(1):80–88, 1985.

[2] T. F. Cox, M. A. A. Cox, and T. F. Cox. *Multidimensional Scaling, Second Edition*. Chapman & Hall/CRC, September 2000.

[3] G. Erich, H. Richard, J. Ralph, and V. John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[4] A. Gray, P. Sallis, and S. MacDonell. Software forensics: Extending authorship analysis techniques to computer programs, 1997.

[5] H. T. Jankowitz. Detecting plagiarism in student pascal programs. *Comput. J.*, 31(1):1–8, 1988.

[6] K.Dauber. *The Idea of Authorship in America*. The University of Wisconsin Press, 1990.

[7] R. Kilgour, A. Gray, P. Sallis, and S. MacDonell. A fuzzy logic approach to computer software source code authorship analysis, 1997.

[8] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 514–524, 1995.

[9] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 514–524, 1995.

[10] S. MicroSystem. Code conventions for the java programming language.

[11] P. W. Oman and C. R. Cook. Programming style authorship analysis. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 320–326, New York, NY, USA, 1989. ACM Press.

[12] W. R. Pestman. *Mathematical Statistics An Introduction*. Walter de Gruyter, 1998.

[13] P. Sallis, A. Aakjaer, and S. MacDonell. Proceedings of the 1996 inerenational conference on software engineering: Education and practice. 1996.

[14] E. H. Spafford and S. A. Weeber. Software forensics: can we track code to its authors? *Comput. Secur.*, 12(6):585–595, 1993.

[15] W.Elliot and R.Valenza. Was the earl of oxford the true shakespeare? *Notes and Queries*, 38:501–506, 1991.

[16] G. Whale. Software metrics and plagiarism detection. *J. Syst. Softw.*, 13(2):131–138, 1990.