Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**
**Corso di Laurea in Informatica**

# TECNICHE DI MACHINE LEARNING PER TYPE INFERENCE DI LINGUAGGI TIPATI DINAMICAMENTE

**Relatore:**
**Chiar.mo prof.**
**Maurizio Gabbrielli**
**Correlatore:**
**dott.ssa Francesca Del Bonifro**

**Presentata da:**
**Bianca Raimondi**

**Sessione II**
**Anno Accademico 2020-21**

# Indice

**1 Introduzione** **1**

**2 Definizione del problema** **3**
2.1 Motivazioni per l'analisi dei tipi nei linguaggi dinamici . . . . 4
2.2 Annotazioni di tipo . . . . . . . . . . . . . . . . . . . . . . . . 5

**3 Tecniche di Machine Learning per type inference** **7**
3.1 DeepTyper . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.2 DLTPy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.3 TypeWriter . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.4 Typilus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.5 LambdaNet . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.6 Type4Py . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.7 HiTyper . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.8 OptTyper . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.9 PyInfer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**4 Dataset** **23**
4.1 Scopo del Dataset . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Struttura del Dataset . . . . . . . . . . . . . . . . . . . . . . 25

**5 Conclusioni** **27**

**Bibliografia** **31**

# Capitolo 1

# Introduzione

Gli sviluppatori spesso preferiscono linguaggi di programmazione tipati dinamicamente, come Python e Javascript, perchè questi linguaggi non richiedono dichiarazioni di tipo esplicite e migliorano la flessibilità dello sviluppatore. Come conseguenza tali linguaggi sono diventati estremamente popolari.
Mentre questo consente una maggiore produttività, la mancanza di tipi all'interno del codice può causare runtime exceptions, inconsistenze di tipo ed è uno dei fattori principali che causano una minor efficienza dell'IDE support, senza considerare il fatto che le compagnie leader di software considerano la tipizzazione statica come una parte importante dello sviluppo di codice.
Risultati di ricerche recenti mostrano i benefici della tipizzazione statica: Gao [1] dimostra che aggiungere annotazioni di tipo in Javascript può aiutare ad evitare un 15% dei bug riportati. Studi precedenti [2] [3] dimostrano anche che sistemi tipati staticamente aiutano nella comprensione di codice privo di documentazione, nel correggere problemi relativi ai tipi e nel risolvere errori di semantica. Infine tutto ciò ha un impatto positivo sul mantenimento del codice.
Nel 2015, Python ha introdotto PEP 484 che consente agli sviluppatori di aggiungere annotazioni di tipo che non comportano cambiamenti a runtime (come ad esempio i commenti): questo permette di conoscere i tipi all'interno di parti di codice che non contengono abbastanza informazioni per essere completamente tipizzate.
Aggiungere in modo statico commenti relativi al codice però comporta un costo. Gli sviluppatori che vogliono utilizzarli hanno bisogno di convertire il loro codice in uno, almeno parzialmente, ricco di annotazioni. Quando la lunghezza del programma risulta molto estesa, aggiungere annotazioni diventa uno sforzo davvero eccessivo.
Data l'importanza di aggiungere informazioni sui tipi all'interno di codice scritto in linguaggi tipizzati dinamicamente, sono state create diverse tecni-

che per attribuire annotazioni in modo automatico. Le tecniche esistenti per l'inferenza di tipo possono essere divise in ampie categorie in base alla loro natura.
In principio sono stati introdotti metodi di analisi statica per l'inferenza di tipo, ma gli stessi sono troppo approssimativi sul comportamento del programma, per questo sono spesso poco precisi. Inoltre questi metodi tipicamente analizzano completamente il programma e le sue dipendenze, motivo per cui risultano molto lenti relativamente a questo task.
Successivamente sono stati creati metodi [4] [5] [6] [7] [8] che risolvessero l'inferenza di tipo su snippets di codice online. Queste tecniche sono state sviluppate e testate per snippets di codice scritti in linguaggi tipizzati staticamente come ad esempio Java. Dato che tali linguaggi sono tipizzati staticamente, gli snippets di codice che vengono presi in considerazione potrebbero non contenere dichiarazioni di tipo e inclusioni di librerie utili alla comprensione dei tipi [6].
Per ovviare alle precedenti limitazioni i ricercatori hanno proposto tecniche basate sul Machine Learning. I risultati sperimentali di questi studi mostrano che le tecniche basate su Machine Learning sono molto più precise dei metodi statici.
L'argomento principale di questa tesi riguarda appunto la descrizione delle principali tecniche di ML presenti in letteratura, le quali verranno illustrate nel capitolo 3.
Per quanto riguarda un problema abbastanza esteso all'interno del Machine Learning, e piú nello specifico per quanto riguarda le tecniche utilizzate per la type inference, risulta essere l'assenza di un dataset per l'addestramento delle reti neurali. Nel capitolo 4 viene mostrata la struttura di un nuovo dataset che in futuro potrebbe essere utile all'addestramento di modelli che siano in grado di analizzare il codice.

# Capitolo 2

# Definizione del problema

I linguaggi di programmazione utilizzano in maniera considerevole tecniche di ingegneria del software, ed é stato mostrato come la scelta di un linguaggio dipenda fortemente dall'opinione che gli sviluppatori hanno del suo design e della sua qualitá. A sua volta, la comunitá accademica ha posto una forte attenzione a valutare l'impatto pratico di importani decisioni di design come la grandezza dei type system e la differenza tra linguaggi statici e linguaggi dinamici.
É stato dimostrato che i linguaggi statici hanno numerosi vantaggi: Hanenberg [2] spiega in un suo studio come i linguaggi tipizzati staticamente abbiano una maggiore manutenibilitá e migliorino la comprensione di codice non documentato, Gao [1] ha scoperto che quando il codice contiene annotazioni di tipo in Javascript é possibile evitare un 15% dei bug riportati.
Nonostante l'importanza dei linguaggi statici, i linguaggi piú diffusi al giorno d'oggi sono tipati dinamicamente: Python, spinto dal suo utilizzo nell'ambiente del Machine Learning, ha raggiunto la prima posizione nella classifica dell'IEEE Spectrum; mentre JavaScript ha ampliato il suo utilizzo grazie allo sviluppo del web per ragioni che includono anche NodeJS.
Raggiungere i benefici della tipizzazione statica in linguaggi dinamici come JavaScript e Python é oggetto di molteplici ricerche.
Molti linguaggi, compreso TypeScript, sono stati sviluppati per proporre un'alternativa migliore: migliorano un linguaggio esistente (come Javascript) con un type system che consente, almeno parzialmente, la tipizzazione statica in modo opzionale permettendo di aggiungere un tipo alle variabili. In questo modo TypeScript puó essere utilizzato e compilato all'interno dell'IDE con tutti i benefici della tipizzazione statica, e puó essere inoltre utilizzato in tutti i contesti in cui viene richiesto codice javascript (vedi Fig. 1.1).
Fortunatamente, lo sviluppo di TypeScript ha permesso una crescita enorme del suo utilizzo; questo ha permesso di creare l'opportunitá di imparare

```javascript
1  function addStyleSheet(ownerDocument,
2                         cssText) {
3      var p = ownerDocument.createElement('p')
4      var parent =
5              ownerDocument.getElementsByTagName('head')[0]
6            || ownerDocument.documentElement;
7
8      p.innerHTML = 'x<style>' + cssText + '</style>';
9      return parent.insertBefore(p.lastChild, parent.firstChild);
10 }
```

```typescript
1  function addStyleSheet(ownerDocument : Document,
2                         cssText : string) : any {
3      var p : HTMLElement = ownerDocument.createElement('p')
4      var parent : HTMLElement =
5              ownerDocument.getElementsByTagName('head')[0]
6            || ownerDocument.documentElement;
7
8      p.innerHTML : string = 'x<style>' + cssText + '</style>';
9      return parent.insertBefore(p.lastChild, parent.firstChild);
10 }
```

Figura 2.1: frammento di codice JavaScript con il corrispondente TypeScript

tecniche di inferenza di tipo attraverso dati dal mondo reale: esso offre un dataset di codici simili a JavaScript con la differenza che contengono anche annotazioni sui tipi, che possono essere convertiti per allenare reti neurali come é stato fatto da Hellendoorn [10].

## 2.1 Motivazioni per l'analisi dei tipi nei linguaggi dinamici

Uno sviluppatore software intento ad editare un file, tipicamente interagisce con diversi tipi di identificatori come nomi di funzioni, parametri e variabili. Ognuno di questi vive nel type system che vincola le operazioni a prendere solo quelli sui quali sono definiti. La conoscenza dei tipi a tempo di compilazione migliora la performance e permette di rilevare a priori vari tipi di errori. Un sistema di tipi efficace é utile anche per tools di sviluppo software come ad esempio la precisione del completamento automatico e le informazioni di debug.

Dato che, almeno virtualmente, ogni linguaggio ha un type system, le informazioni sui tipi possono essere difficili da ricercare a tempo di compilazione senza annotazioni all'interno del codice. Per questo motivo linguaggi come Python e JavaScript sono spesso svantaggiati. Allo stesso tempo, utilizzare annotazioni sui tipi comporta la cosiddetta *type annotation task* [1].

Probabilmente per questo motivo, gli sviluppatori preferiscono usare linguaggi tipati dinamicamente.

Per quel che riguarda Python, l'inferenza di tipo risulta complicata per colpa delle dipendenze da API esterne e delle sue caratteristiche dinamiche. É stato osservato, inoltre, che Python contiene molteplici suggerimenti sui tipi (i cosiddetti *type hints*) come i nomi delle variabili; purtroppo peró questi ultimi non possono essere considerati totalmente affidabili. Le funzioni dichiarate all'interno delle API sono spesso difficili da analizzare a causa dei differenti linguaggi di programmazione usati per dichiararle e la mancanza

di codice sorgente. In aggiunta, le variabili possono avere vari tipi a tempo di esecuzione in base al percorso del programma.

## 2.2 Annotazioni di tipo

I linguaggi tipati dinamicamente non richiedono di inserire manualmente annotazioni sui tipi e i tipi delle variabili vengono chiariti solo a tempo di esecuzione. Essi provvedono ad incrementare la flessibilitá e sono anche molto intuitivi per gli sviluppatori alle prime armi. Purtroppo presentano peró molti inconvenienti: quando un progetto diventa abbastanza esteso potrebbe accadere che neanche uno degli sviluppatori ricordi piú alcune righe di codice. A quel punto, i linguaggi statici possono verificare in automatico la tipizzazione, a differenza di quelli dinamici che richiedono un intervento manuale.
Esiste un dibattito nella comunitá degli sviluppatori su quale tipologia di linguaggi risulti piú idonea al loro utilizzo, ma entrambe le categorie eccellono per diversi aspetti interessanti. C'é un'evidenza scientifica che dimostra che i linguaggi statici forniscano alcuni benefici che evincono al momento in cui un software ha bisogno di essere ottimizzato per migliorarne l'efficienza, la modularitá o la sicurezza. É stata anche data prova per cui tali linguaggi siano meno inclini ad errori rispetto a quelli dinamici.
Linguaggi debolmente tipizzati come JavaScript, PHP e Python non forniscono tali utilitá. Quando questa tipologia di linguaggi viene utilizzata e viene richesta una tipizzazione a tempo di compilazione, sono spesso utilizzate due differenti soluzioni. La prima riguarda l'utilizzo di codice opzionale da incorporare al linguaggio scelto, la seconda invece riguarda l'utilizzo di un linguaggio relativamente molto simile a quello scelto.
Sfortunatamente, le caratteristiche dinamiche di tali linguaggi, come array eterogenei, variabili polimorfe e valutazioni del codice dinamiche rendono l'inferenza di tipo statica un problema molto complicato per linguaggi come Python e JavaScript.
Dato che l'utilizzo di *predictors* probabilistici suggerisce una lista di possibili tipi, scegliere la combinazione del tipo corretto su diversi elementi del programma potrebbe causare un'enorme perdita di tempo. Un approccio naïve potrebbe essere quello di lasciare la scelta del tipo allo sviluppatore. Sfortunatamente questo approccio non é adatto a programmi molto estesi perché il numero di combinazioni da considerare risulterebbe esponenziale nel numero di linee di codice non ancora annotate.
Le annotazioni sui tipi sono solitamente ignorate a runtime, come succede ad esempio per i commenti; tuttavia servono sia come suggerimenti per il

programmatore che utilizza delle API, sia come input dei correttori che assicurano che specifici errori di programmazione non accadano.
Aggiungere annotazioni di tipo a codici scritti utilizzando linguaggi dinamici é diventato abbastanza comune nella moderna ingegneria del software. Python 3.x ha introdotto i type hints tramite il pacchetto *typings* che al giorno d'oggi viene largamente utilizzato assieme al correttore statico mypy [11]. Per JavaScript esistono diverse soluzioni, tra cui TypeScript [12] e Flow [13]. Per gli sviluppatori che desiderano modificare il codice aggiungendo riferimenti per i tipi, come ad esempio in Fig. 1.1, non esiste un tool che possa aggiungerli in modo automatico. Per questo motivo in letteratura si é cominciato a studiare nuove tecniche che lo permettessero, evitando una grande perdita di tempo per lo sviluppatore. Inoltre tali tecniche permettono al sistema di svolgere analisi statiche sul codice quando possibile.
Il fatto di incorporare annotazioni, in termine tecnico *type suggestion task*, che permette di creare codice pienamente annotato é lo scopo di questo lavoro.
Successivamente, nel secondo capitolo, verranno elencate le principali tecniche dello Stato dell'Arte presenti in letteratura.

# Capitolo 3

# Tecniche di Machine Learning per type inference

Come giá introdotto, negli ultimi anni all'interno della comunità accademica si é posta una forte attenzione per tutto ció che riguarda l'inferenza dei tipi a livello di linguaggi dinamici.
Le ultime ricerche riguardano soprattutto linguaggi come Python e JavaScript, e come questi si interfacciano al mondo dell'ingegneria del software. É proprio all'interno di questa branca di studi che i ricercatori cercano di attribuire ai linguaggi dinamici gli stessi benefici posseduti dai linguaggi statici, come ad esempio l'utilizzo di molteplici tool all'interno degli IDE.
Per attribuire gli stessi vantaggi, sono state introdotte diverse tecniche che permettono di aggiungere in modo statico delle annotazioni sui tipi all'interno del codice. Purtroppo la modalitá con cui avviene tale compito comporta un'enorme sforzo e perdita di tempo per lo sviluppatore, il quale preferisce linguaggi dinamici proprio per evitare simili perdite di tempo.
Precisate tali ragioni, nei seguenti paragrafi vengono illustrate le principali tecniche che attribuiscono annotazioni al codice in modo automatico. Queste tecniche fanno uso dell'intelligenza artificiale, e piú nello specifico del Machine Learning, per ovviare a metodi statici che risultano piú lenti e meno precisi. Successivamente vengono elencate e descritte tali tecniche a partire dalle meno recenti fino a concludere con lo Stato dell'Arte.

## 3.1 DeepTyper

DeepTyper é stato introdotto da Hellendoorn [10] nel 2018. Questa tecnica é considerata una dei primi approcci di inferenza di tipo tramite Machine Learning.

DeepTyper é un modello di deep learning che riesce a capire quali tipi occorrono naturalmente in certi contesti e relazioni e riesce a fornire *type suggestion* per codice JavaScript, le quali possono essere verificate da un *type checker* chiamato CheckJS.
DeepTyper fa leva su un insieme di token e tipi automaticamente allineati per predire accuratamente migliaia di annotazioni dei tipi di variabili di funzioni.
Grazie all'utilizzo frequente di TypeScript é stato possibile utilizzare programmi scritti in tale linguaggio per creare un dataset che contenesse codici JavaScript con l'uso di annotazioni. TypeScript, infatti, offre una via di mezzo per JavaScript: TypeScript é un sovrainsieme di JavaScript che offre un type system, il quale permette di ottenere codice parzialmente e facoltativamente annotato che puó essere riconvertito in codice JavaScript.
In tale dataset vengono considerate "*useful type annotation*" le annotazioni di tipo che gli sviluppatori hanno aggiunto manualmente nei progetti scritti in codice TypeScript. Tali annotazioni poi vengono rimosse con il fine di creare un dataset idoneo per JavaScript e attribuirne le giuste *label*.
In TypeScript esistono tre diverse categorie di identificatori che acconsentono all'utilizzo di annotazioni di tipo: i valori di ritorno delle funzioni, i parametri delle funzioni e le variabili.

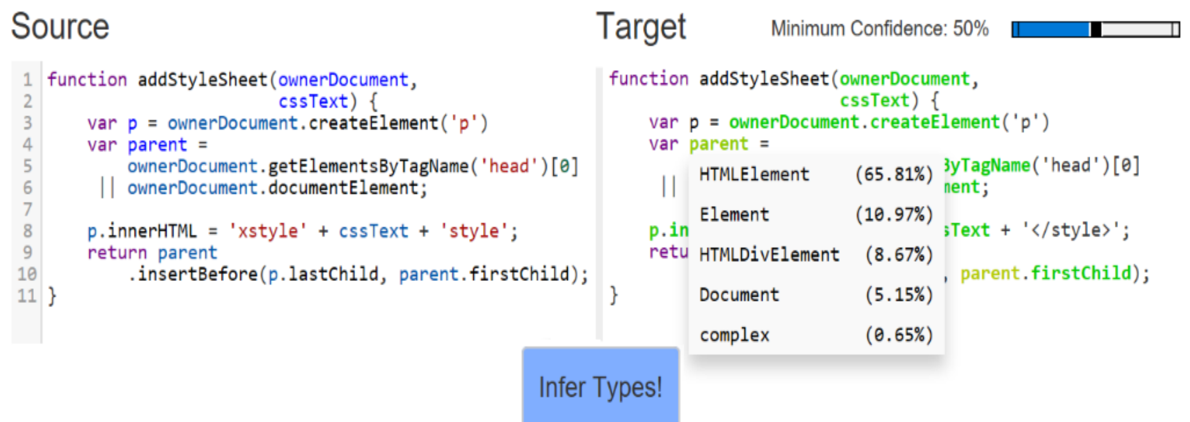



Figura 3.1: Esempio di output di DeepTyper

Il dataset creato viene utilizzato da DeepTyper, il quale si avvale del deep learning su codici realmente esistenti per l'inferenza di nuove annotazioni sui tipi. Questo modello impara ad annotare identificatori con un vettore di tipi: una distribuzione di probabilitá sui tipi viene usata per proporre tipi che CheckJS puó verificare e rilasciare all'utente come *type suggestion* (vedi

Figura 3.1). Per ottimizzare il training, non si attribuiscono solo annotazioni nella posizione dove viene richiesta la dichiarazione ma anche in tutte le occorrenze comprese all'interno del codice. Questo fa sí che il deep learner faccia uso di un contesto molto piú ampio.
Nello stesso articolo viene dimostrato come il contesto sia la chiave che porta a predire il tipo nel modo piú accurato possibile. Avere a disposizione un contesto molto vasto risulta cruciale nel predire l'annotazione di tipo di una variabile, questo perché permette al modello di imparare da dipendenze che abbiano statisticamente un intervallo ampio, come ad esempio la distanza che occorre tra il tipo di una funzione ed il tipo del suo valore di ritorno. Sempre per quanto riguarda il contesto, includere gli identificatori (ad esempio i nomi delle variabili), permette al modello di incorporare informazioni probabilistiche sui nomi al problema di ottenere il giusto tipo come *type suggestion*.
L'idea di base di DeepTyper si basa sull'utilizzo del NLP (*Natural Language Processing*) per conoscere il ruolo di un elemento all'interno di una frase e di conseguenza intuirne il tipo. Ad esempio, la parola 'mail' puó essere sia un verbo che un soggetto, ma dato il contesto, nella frase 'i will mail the letter tomorrow' il suo utilizzo diventa scontato.
In questo articolo viene dimostrato inoltre come il problema riguardante l'inferenza di tipo possa essere risolto da tecniche di Machine Learning: tali tecniche arricchiscono la type inference convenzionale utilizzando informazioni molto importanti ed utili basate sui nomi delle variabili e sul contesto. Questo permette di considerare il Machine Learning una tecnica ideale per questo compito.

## 3.2 DLTPy

Similmente a DeepTyper nel 2019 é stato introdotto DLTPy (Deep Learning Type inference of Python elements) [14].
DLTPy uilizza una tecnica molto simile a Deeptyper con la differenza che il training viene effettuato su codice scritto in linguaggio Python.
Questa tecnica si basa soprattutto sull'utilizzo del contesto: commenti, elementi semantici relativi al nome delle funzioni e ad i loro parametri e identificatori del valore di ritorno. Utilizzare il linguaggio naturale di questi elementi permette di allenare un classificatore che predica i tipi nel modo piú corretto.
Similmente a DeepTyper il modello viene allenato utilizzando progetti open source che contengono annotazioni di tipo; durante il training queste annotazioni vengono rimosse e utilizzate per attribuire le corrispondenti label. Tale tecnica funziona perché il codice risulta ripetitivo e intuitivo.

Boone [14] all'interno della sua ricerca ha assunto che i commenti e i nomi degli identificatori fossero il punto fondamentale della funzione. Inoltre considerare anche le espressioni di ritorno apporta valore alla precisione del modello e e si ipotizza che includere commenti abbia un effetto molto positivo sui risultati.

## 3.3 TypeWriter

TypeWriter, introdotto in Marzo 2020 da Pradel [15], é la prima tecnica di inferenza di tipo per Python che combina predizioni probabilistiche sui tipi assieme a NLP.
Tale modello impara a riconoscere il tipo del valore di ritorno e degli argomenti delle funzioni partendo da codice parzialmente annotato e combinando le proprietá del linguaggio naturale all'interno del codice con informazioni sul livello del linguaggio di programmazione.

```
1  # Predicted argument type: int, str, bool
2  # Predicted return type: str, Optional[str], None
3  def find_match(color):
4    """
5    Args:
6      color (str): color to match on and return
7    """
8    candidates = get_colors()
9    for candidate in candidates:
10     if color == candidate:
11       return color
12   return None
13
14 # Predicted return types: List[str], List[Any], str
15 def get_colors():
16   return ["red", "blue", "green"]
```

Figura 3.2: Esempio di output di TyperWriter

L'input di TypeWriter é un insieme di linee di codice dove alcuni, ma non tutti i tipi sono stati annotati. L'approccio é diviso in tre parti principali. Inizialmente, un'analisi statica veloce estrae alcuni tipi di informazioni dal codice dato. Le informazioni estratte includono informazioni strutturali del programma, come ad esempio l'utilizzo degli argomenti delle funzioni, e informazioni sul linguaggio naturale, come ad esempio i nomi degli identificatori e i commenti. Nella seconda parte un *neural type predictor* impara da tipi giá annotati e dalle informazioni ad essi associate come predire i tipi

rimanenti all'interno del codice. Una volta allenata la rete, il modello é in grado di predire tipi simili per le parti del codice non ancora annotate. In fine una ricerca *deefback-directed* utilizza il modello allenato per trovare assegnamenti di tipo che siano consistenti e corretti in accordo con un *type checker* statico. L'output risultante di TypeWriter é codice contenente annotazioni aggiuntive. Un esempio dell'output di questo modello puó essere osservato in Figura 3.2.
Per quanto riguarda l'analisi statica iniziale, essa raccoglie quattro tipi di informazioni sul contesto. Ognuno di essi fornisce suggerimenti riguardo il tipo di un argomento o del valore di ritorno della funzione, e il modello é in grado di combinare tali suggerimenti con predizioni sul tipo piú probabile. I quattro tipi di informazioni di contesto sono: i nomi degli identificatori che richiedono l'aggiunta di annotazione, le occorrenze dell'elemento all'interno del codice, i commenti alla funzione e i tipi disponibili al momento (dei quali la rete é a conoscenza grazie alle dichiarazioni di tipo presenti ed alle librerie importate). Per ció che riguarda i nomi degli identificatori, risultano utili perché da essi é molto probabile intuirne il tipo: ad esempio un elemento chiamato "name" é molto probabile che sia di tipo stringa, mentre uno nominato "doPropagate" é probabile che sia un booleano.
TypeWriter é in grado di predire solo i tipi che sono parte del suo vocabolario. Quando la grandezza del vocabolario viene configurata ad utilizzare 1000 tipi, si puó effettuare con una stima che il 90% dei tipi che verranno richiesti sia all'interno del vocabolario. Tuttavia, dato l'evolversi del software, gli sviluppatori creano nuovi tipi o cambiano i nomi a quelli giá esistenti. Questo potrebbe creare situazioni in cui il modello predica tipi sbagliati perché i nomi vengono modificati o perché semplicemente non é a conoscenza della loro esistenza. Tale problema riguardante il vocabolario é molto studiato in ingegneria del software, e TypeWriter potrebbe utilizzare una delle tecniche proposte per migliorare questo problema. In ogni caso, messo a confronto con un tool di inferenza di tipo statico, TypeWriter aggiunge alcuni tipi al vocabolario.
TypeWriter oltre ad essere una tecnica di type inference molto utile, contribuisce soprattutto, in modo significativo, alla precisione ed alla velocitá delle tecniche probabilistiche di type inference. In questo senso tecniche come DeepTyper potrebbero venire combinate con tali scoperte per migliorare le proprie caratteristiche in questo ambito.
In conclusione questo modello viene attualmente utilizzato e testato da sviluppatori di Facebook che ne fanno uso apportando solo piccole modifiche alle annotazioni riportate.

## 3.4 Typilus

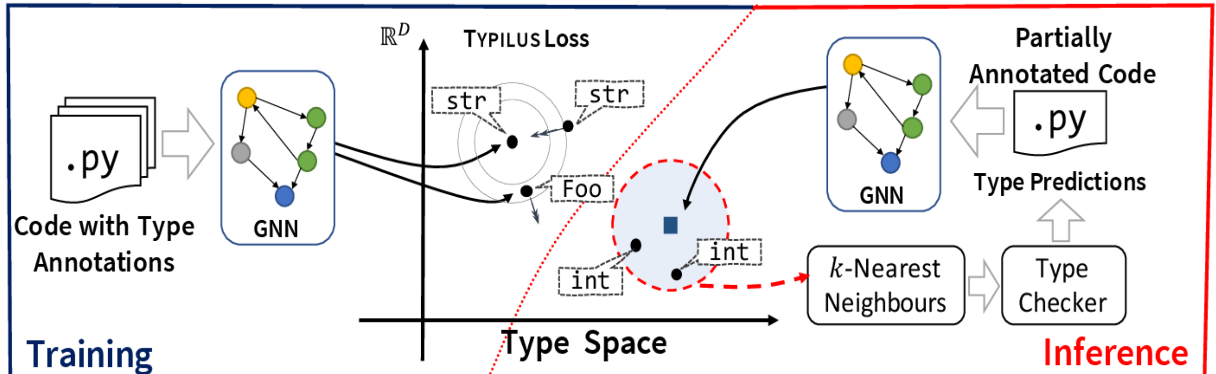



Figura 3.3: Esempio di utilizzo di Typilus

Typilus, introdotto da Allamanis nel 2020 [9], é un modello GNN (*Graph Neural Network*) che predice i tipi ragionando sulla struttura del programma, sui nomi e sui patterns.
Questa rete utilizza il *deep similarity learning* per creare un *TypeSpace*. Inoltre questo modello utilizza la tecnica del *one-shot learning* per creare un vocabolario dei tipi includendo anche quelli piú rari e quelli definiti dall'utente.
Typilus ha fatto progredire lo Stato dell'Arte formulando type inference probabilistica come un problema che basa la sua metrica sul *meta-learning*. Il *meta-learning*, detto anche *learning-to-learn*, permette ai modelli di machine leraning allenati di adattarsi a nuove impostazioni utilizzando un singolo esempio e senza training aggiuntivo.
In contrasto con metodi di classificazione, Typilus puó predire efficientemente i tipi non ancora conosciuti durante il training. Una volta allenata la rete, essa puó essere usata per calcolare la rapprsentazione di un simbolo di un nuovo tipo nel *TypeSpace*. Typilus conserva una mappa dei tipi che rappresenta il loro valore *embedding*. Questa mappa implicitamente definisce le "regioni dei tipi" nel TypeSpace. In questo modo, all'occorrenza di un nuovo tipo, la rete é in grado di predire il suo tipo in base alla vicinanza dello stesso all'interno della mappa. Per questo motivo Typilus puó supportare un vocabolario *open type* senza *retraining*.
Typilus é stato realizzato utilizzando una GNN (*Graph Neural Network*). Molti approcci che utilizzano reti neurali trattano i programmi come sequenze di token ed altri invece utilizzano librerie che trasformino il codice in un AST (albero di sintassi astratta), tralasciando l'opportunitá del modello di poter imparare dalle complesse dipendenze tra i token all'interno del codice. Allamanis [9] dimostra come utilizzare un modello GNN produca un 7.6% di

precisione in piú rispetto all'utilizzo di altre tecniche per quanto riguarda i tipi piú comuni.
Assieme al TypeSpace, Typilus utilizza anche un *type checker* come riportato in Figura 3.3.
Diversamente dagli altri modelli, quest'utlimo riesce anche a scoprire annotazioni sui tipi incorrette; questo ha permesso a due librerie importanti e abbastanza conosciute di variare il loro codice aggiungendo i commit derivanti dall'utilizzo di Typilus sul loro codice.

## 3.5 LambdaNet

```
1   var c1: τ8 = class MyNetwork {
2     name: τ1;  time: τ2;
3     var m1: τ9 = function forward(x: τ3, y: τ4):τ5 {
4       var v1: τ10 = x.concat;  var v2: τ11 = v1(y);
5       var v3: τ12 = v2.TIMES_OP; var v4: τ13 = v3(NUMBER);
6       return v4;
7     }
8   }  // more classes...
9   var f1:τ14 = function restore (network: τ6): τ7 {
10    var v3: τ15 = network.time;
11    var v4: τ16 = readNumber(STRING);
12    network.time = v4;  // more code...
13  }
```

Figura 3.4: Esempio di codice sorgente annotato da LambdaNet

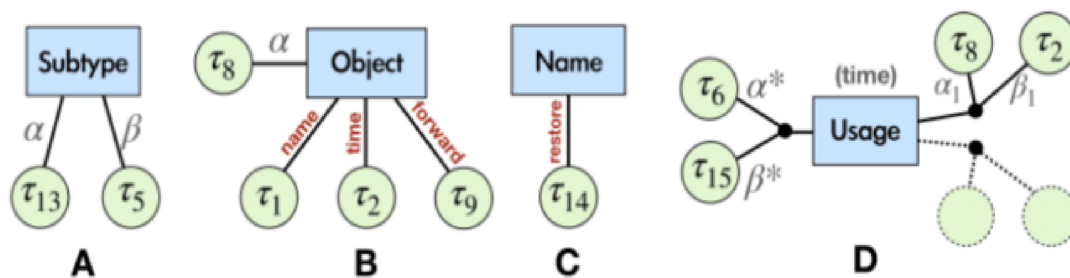


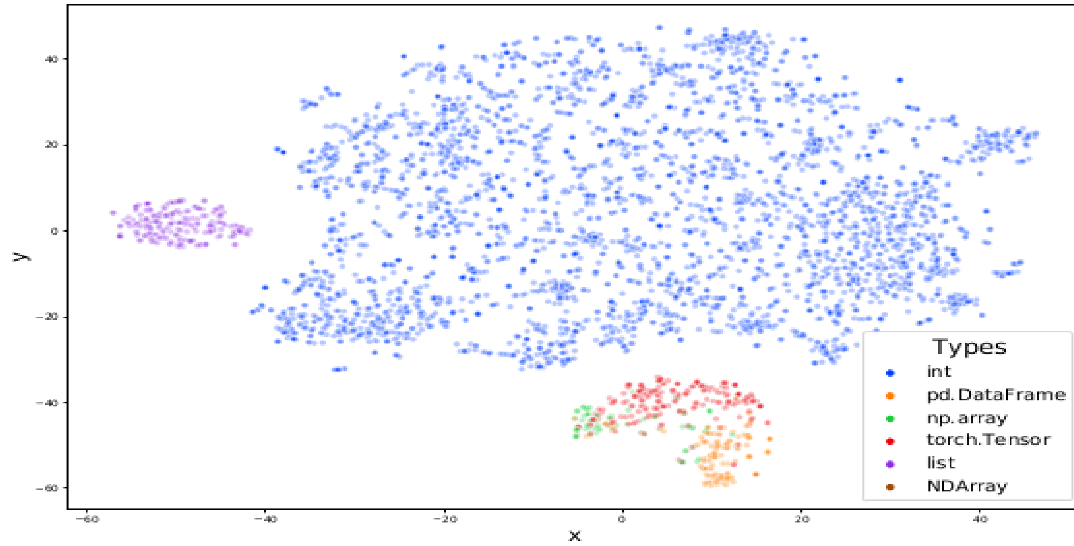

Figura 3.5: Esempio di grafo creato a partire dal sorgente in Figura 3.4

Il corrispettivo di Typilus per TypeScript é stato introdotto da Wei [16] nel 2020.
Nel suo articolo Wei propone un nuovo algoritmo probabilistico per la type inference facendo uso di una GNN, come appunto per Typilus.

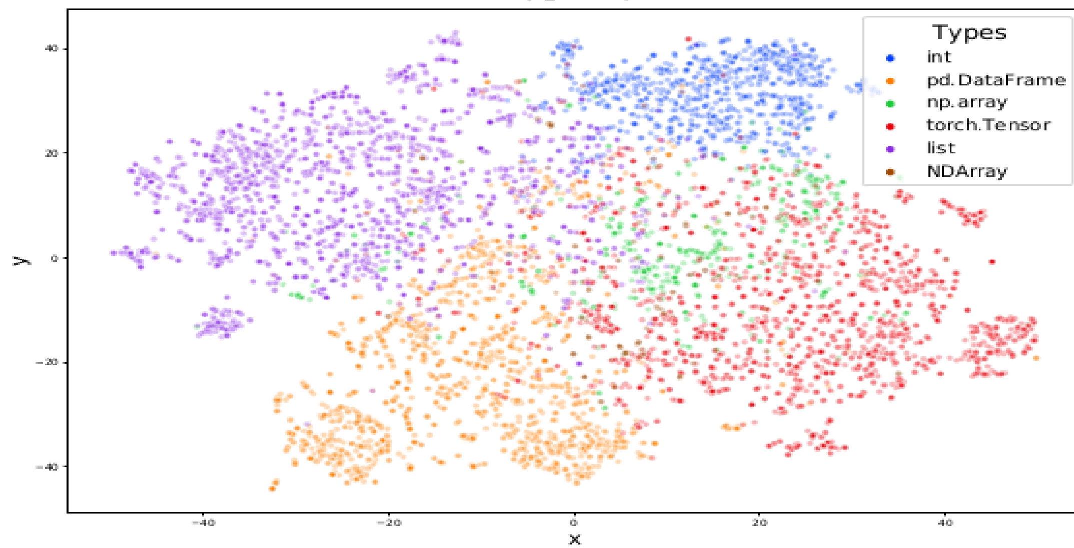
Tale metodo si avvale di analisi su codice sorgente per trasformare un programma in una nuova versione rappresentata da un grafo, dove i nodi rappresentano i tipi delle variabili e le etichette degli iperarchi codificano le relazioni tra essi. Inoltre per esprimere i vincoli logici come nella tradizionale type inference, il grafo delle dipendenze sopra descritto incorpora anche suggerimenti contestuali che comprendono l'utilizzo di nomi e variabili. Un esempio di tale tecnica viene descritto in Figura 3.5.
Come per Typilus esistono diversi metodi per rappresentare un programma: sequenze di token, alberi di sintassi astratta, grafi di controllo del flusso, etc.. Tuttavia, nessuna di queste codifiche risulta particolarmente utile per trovare le corrette annotazioni di tipo. Per questo motivo, LAMBDANET utilizza un'analisi statica per dedurre un insieme di predicati che risultano rilevanti per il problema dell'inferenza di tipo e rappresenta questi predicati facendo uso di astrazioni del programma chiamate *type dependency graph*.

## 3.6 Type4Py

Type4Py é stato introdotto da Mir [17] nel 2021.
É un modello per l'inferenza di tipo di codice Python basato su deep similarity learning. Questo modello impara a distinguere tipi diversi e li colloca in uno spazio multidimensionale, il quale divide fisicamente tipi differenti. Formulando il problema della type inference come un problema di Deep Similarity Learning (DSL), viene proposta un'effettiva rete neurale gerarchica (HNN) che é in grado di mappare i programmi in uno spazio multidimensionale in base alle proprie caratteristiche, anche conosciuta con il nome di *type cluster*. La tecnica del similarity learning viene usata spesso per problemi riguardanti Computer Vision, ad esempio per riconoscere il viso di una persona tra tanti altri. Similmente, Type4Py impara a distinguere i tipi degli elementi all'interno del codice utilizzando una HNN.
Diversamente dalla tecnica precedente, Type4Py estrae i *type hints* dal codice utilizzando un AST. Da esso é possibile ottenere suggerimenti riguardanti il tipo del valore di ritorno e degli argomenti delle funzioni. Come per gli approcci antecedenti, tali suggerimenti si basano sul contesto e sul linguaggio naturale.
Nello stesso articolo, Mir [17] espone il fatto che utilizzare il giusto dataset sia uno dei principali fattori per migliorare l'efficienza del modello. Di fatto, nel caso della type inference per Python, utilizzare un dataset che faccia il piú possibile uso di annotazioni risulta la soluzione migliore. Per questo motivo Mir ha creato un dataset utilizzando il servizio web libraries.io che in questo caso viene utilizzato per ricercare i packages che includono *mypy*, il




(a) Type4Py




(b) Typilus

Figura 3.6: Confronto tra Type4Py e Typilus

*type checker* piú utilizzato per python la cui funzione é basata sulla presenza di annotazioni.
Alcune ricerche hanno mostrato che gli sviluppatori che fanno uso di tecniche automatiche per l'aggiunta di annotazioni sui tipi preferiscono utilizzare il primo suggerimento derivante da tali tecniche. Di conseguenza risulta importante che le tecniche basate sul Machine Learning mostrino risultati soddisfacenti nella *top-n prediction*, specialmente nella *top-1*. La precisione per quel che riguarda Type4Py, é stata migliorata a confronto di tecniche simili come Typilus o TypeWriter.
Mir in conclusione ha dimostrato come il nome degli identificatori delle variabili all'interno del programma sia il maggiore suggerimento da cui si possono trarre type hints riguardanti il tipo della variabile.
All'interno di questo studio Type4Py viene paragonato alla tecnica di Typilus. In Figura 3.6 é possibile distinguere i due diversi approcci elaborati sullo stesso dataset di Type4Py. Si puó vedere come i cluster in Type4Py siano ben separati e piú compatti rispetto a quelli in Typilus.

## 3.7 HiTyper

```
1  #src/graph_transpiler/webdnn/graph/shape.py
2  def parse(text):
3      normalized_text = _normalize_text(text)
4      tmp = ast.literal_eval(normalized_text)
5      shape = []
6      placeholders = {}
7      for i, t in enumerate(tmp):
8          if isinstance(t, str):
9              pt = Placeholder(label=t)
10             placeholders[t] = pt
11         elif isinstance(t, int):
12             pt = t
13         shape.append(pt)
```

Figura 3.7: Esempio di codice in input per HiTyper

HiTyper [18] é un framework per l'inferenza di tipo ibrido: integra inferenza statica assieme ad un modello di deep learning.
Diversamente dalle tecniche precedenti, HiTyper viene utilizzato per predire, oltre agli argomenti ed ai valori di ritorno delle funzioni, anche le variabili, le quali possono risultare utili per vari task come rappresentazione del codice, raccomandazioni di API e rilevazione di errori.

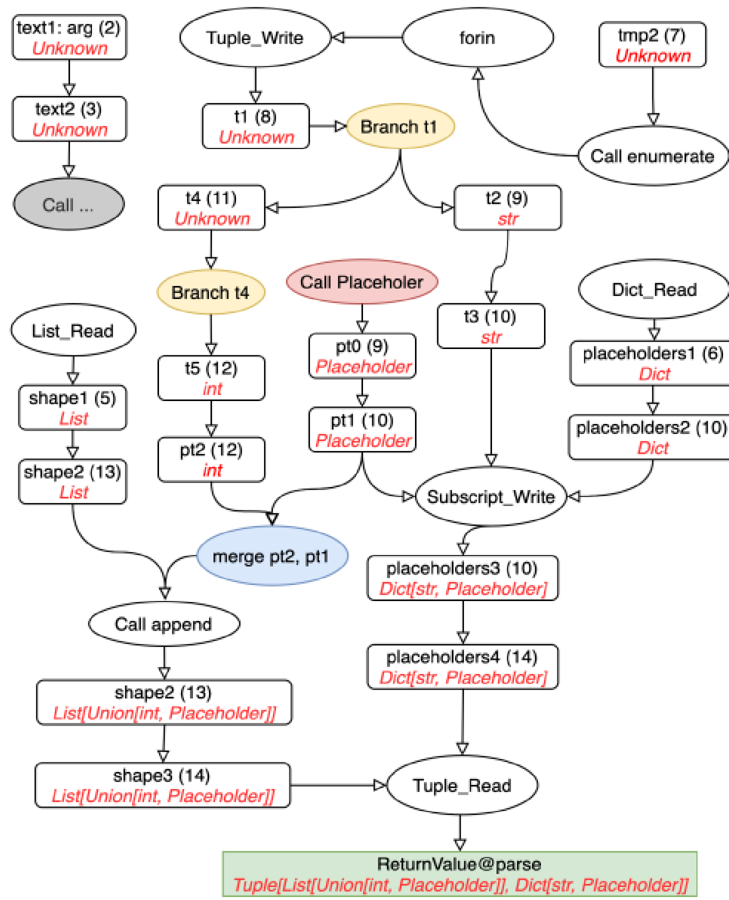



Figura 3.8: Esempio di *type graph* relativo alla Figura 3.7

HiTyper accetta come input un codice sorgente scritto in Python e rilascia come output un file JSON contenente gli assegnamenti di tipo. La sua architettura é composta da tre maggiori componenti: la generazione del grafo dei tipi (*type graph*), l'utilizzo di tecniche statiche e dinamiche per la type inference e la predizione del tipo.
Il grafo dei tipi viene creato per ogni funzione presente all'interno del codice assieme ai tipi definiti dall'utente. Successivamente molti elementi vengono tipati grazie ad analisi statiche sul grafo, i rimanenti vengono esaminati utilizzando una tecnica che utilizza ML per il fatto che tali elementi non presentano abbastanza vincoli statici per poter essere predetti con l'analisi statica precedente.
All'interno del grafo, i nodi rappresentano tutte le variabili e le operazioni nel codice sorgente, mentre gli archi rappresentano la provenienza di un tipo a partire dal nodo padre. In tale grafo viene rappresentato il flusso dei tipi, ovvero per ogni variabile viene rappresentata con un nodo ogni sua occorrenza, in modo tale che, dato che le variabili nei linguaggi dinamici possono cambiare tipo a runtime, sia possibile accertarsi del tipo ad ogni sua occorrenza.
Per allenare la rete, Peng ha utilizzato lo stesso dataset di Typilus. Questo dataset contiene un insieme di progetti derivanti da GitHub in modo tale da allenare la rete utilizzando codici realmente utilizzati nel modo reale. Tuttavia, la distribuzione dei tipi puó variare in base ai progetti scelti, inoltre l'utilizzo di annotazioni di tipo non é utile per quel che riguarda le variabili, ma solo per gli argomenti e i valori di ritorno delle funzioni. Tale problema limita la performance dei modelli di ML in questo ambito. Come ricerca futura bisognerebbe creare un dataset opportuno e renderlo disponibile per tali scopi.

## 3.8 OptTyper

Pandi [19] nel 2021 ha introdotto OptTyper, una tecnica per TypeScript che combina vincoli logici, ovvero informazioni deterministiche provenienti dal type system, e vincoli naturali, ovvero informazioni statistiche non certe su tipi riconosciuti dal sorgente come i nomi degli identificatori.
I vincoli logici vengono estratti direttamente dal codice e il deep learning viene applicato per predire i tipi da proprietá del codice ad un livello superficiale. I vincoli logici risultano di grande importanza perché come dimostrato non avrebbe senso suggerire tipi che violino vincoli sui tipi.
Pandi dimostra che OptTyper risulta molto piú performante rispetto ad approcci che utilizzano solamente il deep learning.

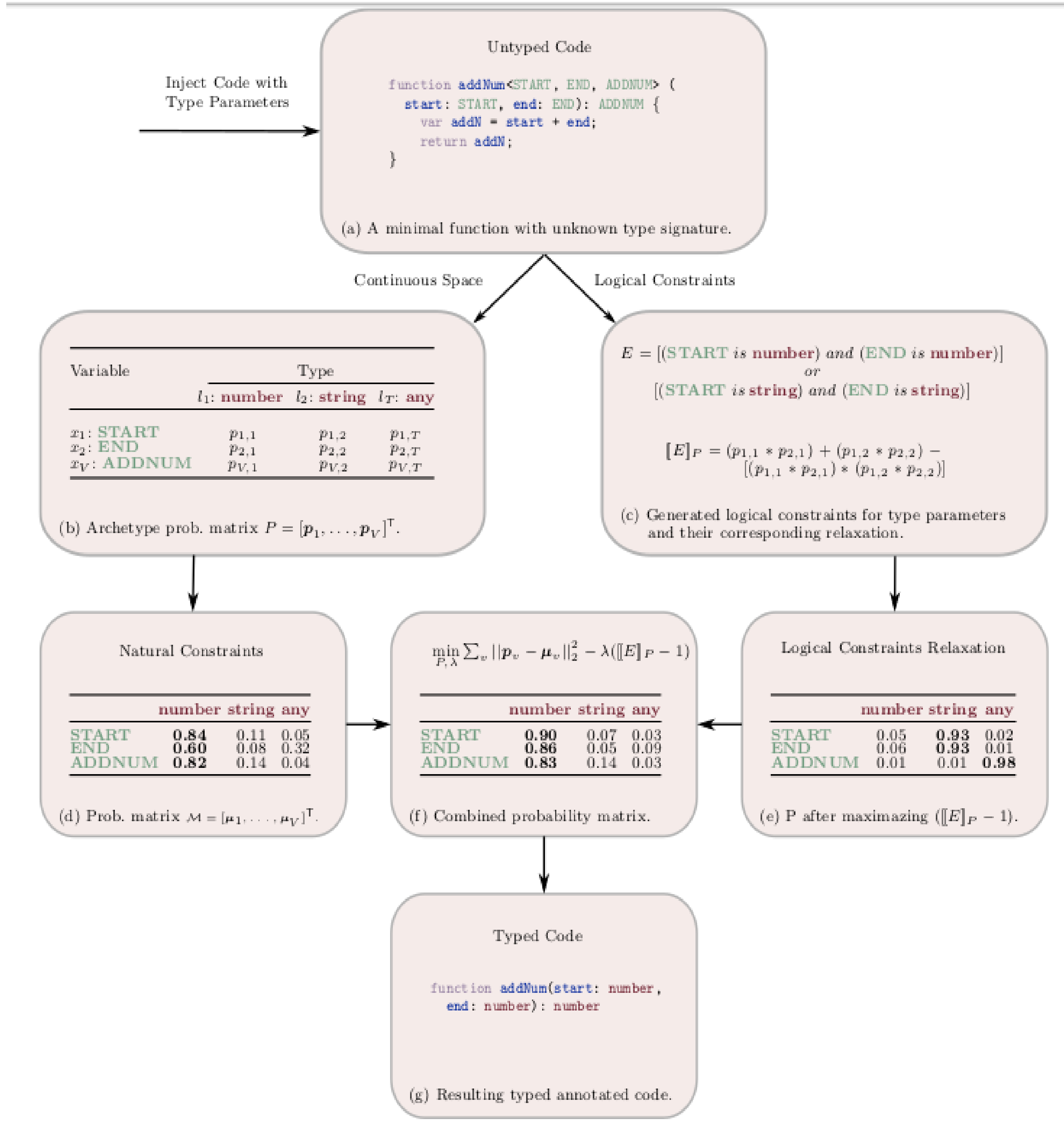



Figura 3.9: Esempio di OptTyper su codice

La Figura 3.9 illustra il comportamento generale di OptTyper grazie ad un semplice esempio dove vengono predetti dei tipi. L'input é una funzione senza annotazioni di tipo sui parametri o valori di ritorno. Nel Box (a) viene associato ed inserito inizialmente un tipo per ogni parametro e uno per il valore di ritorno (START, END, ADDNUM). I vincoli logici riguardo tali tipi rappresentano la conoscenza ottenuta da analisi simboliche del codice all'interno del corpo delle funzioni. In questo esempio, l'utilizzo di un'operazione binaria implica che i tipi dei due parametri dell'operazione sono uguali. Il box (c) mostra un piccolo insieme di vincoli logici che rappresentano il fatto che i due parametri della funzione addNum hanno lo stesso tipo. Il problema é che avendo solo vincoli logici non é possibile fare un'approssimazione sul tipo: non é possibile riconoscere se il tipo sia un numero o una stringa e questo potrebbe portare il modello a classificare i parametri come "any". L'apporto cruciale di OptTyper é quello di tenere conto anche di vincoli del linguaggio naturale, ovvero vincoli riguardanti i nomi degli identificatori e come essi vengano interpretati da intenzioni umane. In particolare vengono applicate tecniche di ML che imparano da un grosso insieme di dati. Grazie a tale tecnica il box (d) mostra i vincoli naturali indotti dai nomi degli identificatori, questo porta a creare dati relativi a probabilitá che possono essere utilizzate assieme ai vincoli logici.

## 3.9 PyInfer

PySonar2

```
range_header = request.headers.get("Range")
#  range_header:?
def media_endpoint(_id): # media_endpoint:? -> ? | _id:?
    if range_header:
        status = 206 #  status:int
        size = file_.length
        try:
            m = re.search(r"(\d+)-(\d*)", range_header)
            # m:?
            begin, end = m.groups()
            #  begin:? | end:? | m:?
            begin = int(begin)
            #  begin:int
            end = int(end)
            #  end:int
        except:
            begin, end = 0, None #  begin:int | end:None
        length = size - begin
        #  length:int
        if end is not None:
            length = end - begin + 1
            #  length:int
        file_.seek(begin)
```

PYInfer

```
range_header = request.headers.get("Range")
#  range_header:str - 0.0232 | request:str - 0.1303
def media_endpoint(_id):
    if range_header: #  range_header:str - 0.3134
        status = 206 #  status:int - 0.9932
        size = file_.length #  file_:int - 0.7147 | size:int - 0.8966
        try:
            m = re.search(r"(\d+)-(\d*)", range_header)
            #  m:str - 0.4095 | range_header:int - 0.5748 | re:str - 0.4905
            begin, end = m.groups()
            #  begin:int - 0.9219 | end:int - 0.9441 | m:int - 0.6654
            begin = int(begin)
            #  begin:int - 0.9962 | begin:int - 0.8097 | int:int - 0.9914
            end = int(end)
            #  end:int - 0.9962 | end:int - 0.7749 | int:int - 0.9893
        except:
            begin, end = 0, None #  begin:int - 0.9299 | end:int - 0.9954
        length = size - begin
        #  begin:int - 0.6510 | length:int - 0.9854 | size:int - 0.9354
        if end is not None: #  end:int - 0.8745
            length = end - begin + 1
            #  begin:int - 0.8548 | end:int - 0.8731 | length:int - 0.9959
        file_.seek(begin) #  begin:int - 0.8135 | file_:str - 0.6348
```

Figura 3.10: Confronto tra PySonar e PyInfer

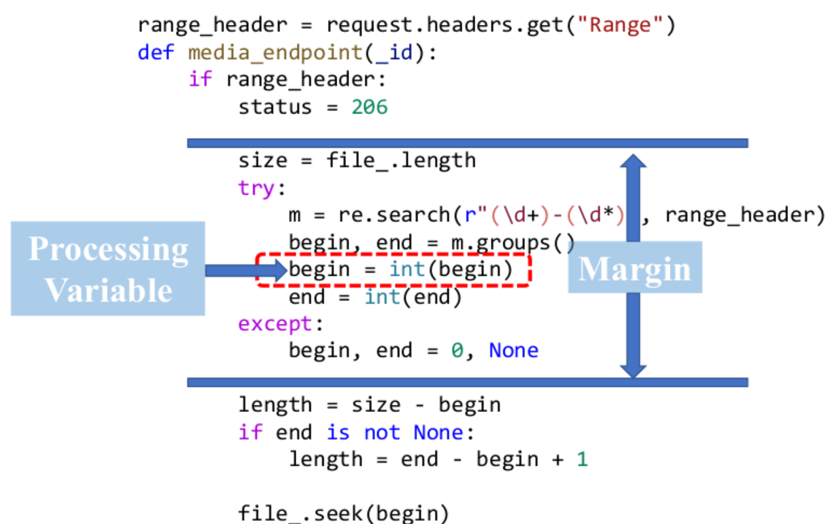



Figura 3.11: Margine semantico per le variabili per PyInfer

PyInfer [20], come le precedenti tecniche, si basa sul deep learning per generare annotazioni di tipo per Python.
Dato che risulta complicato trovare un dataset di progetti Python con codice giá annotato, Cui nel suo elaborato utilizza un analizzatore statico, PySonar2 [21], per generare, almeno parzialmente, annotazioni in automatico. In un secondo momento vengono applicate tecniche per raffinare la qualitá del dataset.
Un confronto con l'analizzatore statico PySonar2 viene raffigurato in Figura 3.10, dove PyInfer impiegando un tempo molto simile é in grado di predire annotazioni in quantitá 5 volte superiore a PySonar2.
Uno dei punti principali di PyInfer é quello di fare leva su aspetti del codice semantici per l'inferenza di tipo delle variabili. Cui [20] ipotizza che il contesto ottenuto a partire da un certo margine definito rispetto all'occorrenza della variabile sia un'informazione semantica rilevante che caratterizza la stessa. In questo modo PyInfer é capace di analizzare la semantica delle variabili assieme alla sintassi strutturale e ad informazioni relative alla grammatica del linguaggio. In Figura 3.11 viene rappresentata l'impostazione del margine per la variabile presa in considerazione. Per ogni variabile vengono memorizzati i token del codice sorgente assieme ai loro scopi contestuali. Questo permette di fornire informazioni contestuali per le variabili che risultano utili per predire le annotazioni relative
PyInfer fornisce annotazioni di tipo in un millisecondo per una variabile. Fornisce dunque annotazioni sui tipi in concomitanza della loro stesura. PyInfer puó essere usato anche per rilevare i tipi degli argomenti delle funzioni. Da-

to che, diversamente da tecniche illustrate in precedenza, questo approccio utilizza semantiche di alto livello al posto di grafici, puó essere facilmente esteso per annotare variabili e determinare errori di semantica in qualsiasi linguaggio di programmazione.

# Capitolo 4

# Dataset

Ultimamente sono state compiute molte attivitá di ricerca nell'ambito dell'analisi del codice sorgente, sia per quanto riguarda l'ingegneria del software che per quanto riguarda il Machine Learning. Parlando di ingegneria del software le analisi compiute sono servite maggiormente da appoggio per creare nuove tecniche di sviluppo e di manutenibilitá del codice. Per quel che riguarda il machine learning, invece, come illustrato nei capitoli precedenti, é stata posta una maggiore attenzione al codice e piú nello specifico al miglioramento delle tecniche di type inference, soprattutto per ció che concerne l'utilizzo di linguaggi dinamici come Python e JavaScript.
In questo ambito sono state compiute analisi che riportano conseguentemente ad argomenti di ingegneria del software, come l'utilizzo di tool automatici per l'aggiunta di annotazioni all'interno del codice.
Per quanto riguarda la letteratura di riferimento, uno dei principali problemi inerenti le tecniche di analisi del codice che utilizzano modelli di ML, é a tutti gli effetti l'assenza di dataset che possano essere utilizzati per allenare i modelli creati. Questo capitolo si interessa dunque della creazione di un dataset idoneo allo sviluppo di modelli all'interno dell'ambiente dell'intelligenza artificiale applicata alla type inference.

## 4.1 Scopo del Dataset

Il seguente dataset é composto da un insieme di progetti principalmente derivanti da GitHub. Tali progetti sono composti da file scritti in Python: la scelta di questo linguaggio é stata fatta per l'ovvia utilitá nel mondo della type inference di linguaggi dinamici.
Python é stato scelto, come da molte tecniche esposte nel capitolo precendente, per l'ampio utilizzo che se ne fa per creare reti neurali. Ultimamente

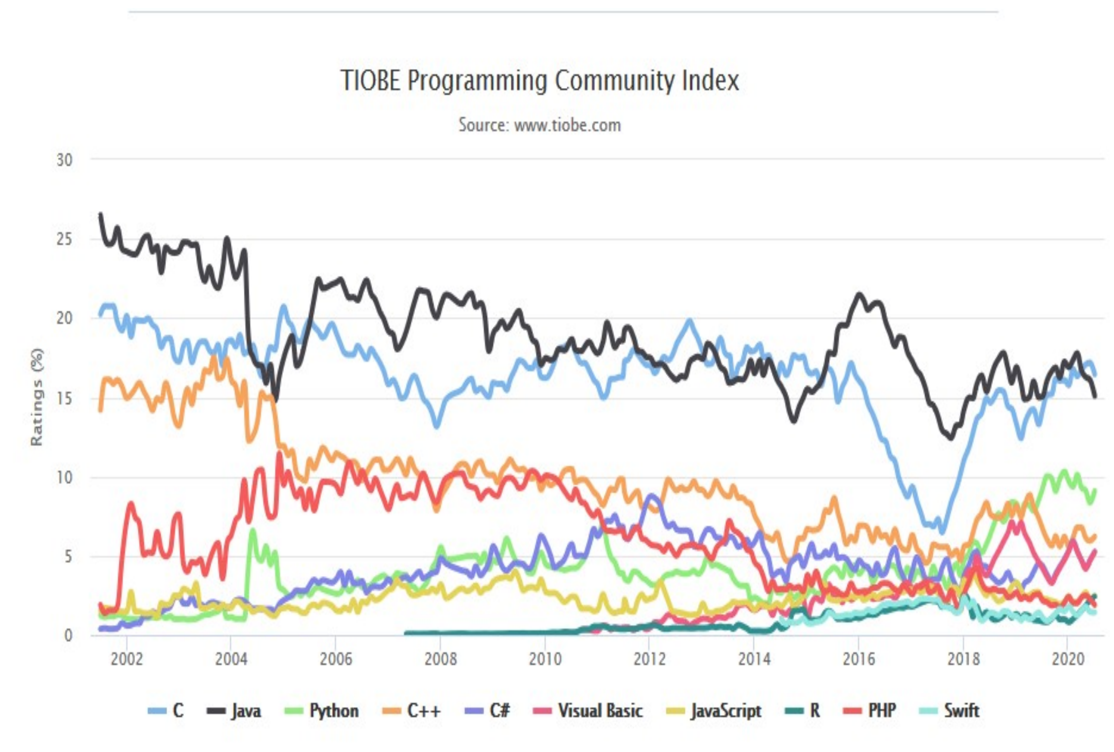



Figura 4.1: Utilizzo dei principali linguaggi di programmazione nel mondo reale

tale linguaggio é uno dei maggiori utilizzati in generale, e non solo all'interno di tecniche finalizzate all'intelligenza artificiale: in particolare risulta essere il terzo maggior linguaggio in uso. Gli unici linguaggi utilizzati piú di frequente rispetto a Python risultano essere C e Java, ma essi, come mostrato in Figura 4.1, nel corso del tempo stanno pian piano diminuendo il loro vantaggio a favore di un forte incremento di Python.
Un futuro interesse potrebbe essere destinato alla ricerca di progetti che contengano codice JavaScript, non tanto per la sua popolaritá, in quanto risulta nella media, ma quanto per il fatto che possa essere scelto come linguaggio dinamico su cui svolgere analisi relative alla type inference.

## 4.2 Struttura del Dataset

Prendendo spunto da Mir [17], che nel suo modello utilizza una collezione di progetti Python, all'interno del dataset proposto sono presenti progetti che facciano ipoteticamente uso di annotazioni. Come illustrato nel suo articolo, infatti, Mir ipotizza come progetti che utilizzano librerie di analizzatori statici siano tali da garantire che nel codice siano presenti annotazioni. Il motivo di tale supposizione riguarda ad esempio l'utilizzo dell'analizzatore mypy per quanto riguarda Python: tale analizzatore per verificare errori o inconsistenze di tipo all'interno del codice ha bisogno di annotazioni per quanto riguarda soprattutto i tipi, senza le quali non sarebbe capace di esaminare il programma.
Utilizzando la precedente idea é stato possibile verificare le dipendenze dei progetti e scaricarli da GitHub.
L'insieme dei progetti di cui é stato fatto il download é composto da 597 cartelle, ognuna contenente un progetto differente. Da ogni cartella sono stati poi eliminati tutti i file che non avessero estensione in python (ad esempio file di librerie e dipendenze utilizzate all'interno del codice python) in quanto non rilevanti per l'analisi del programma.

```
def is_local_file(obj: Optional[Union[str, Path]]) : -> bool
    """
    Checks if a given string is a file on local system.

    Args:
        obj (:obj:`str`): The string to check.
    """
    if obj is None:
        return False

    path = Path(obj)
    try:
        return path.is_file()
    except Exception:
        return False
```

```
def is_local_file(obj: Optional[Union[str, Path]]) :
    """
    Checks if a given string is a file on local system.

    Args:
        obj (:obj:`str`): The string to check.
    """
    if obj is None:
        return False

    path = Path(obj)
    try:
        return path.is_file()
    except Exception:
        return False
```

A partire dal dataset creato, é stato realizzato un nuovo dataset contenente

9 cartelle: una per ogni tipo statico utilizzato in python per le annotazioni. Tali cartelle sono state ridenominate in base al tipo: str, bool, int, float, dict, list, tuple e set. In ogni directory sono stati aggiunti file, ognuno contenente il codice di una funzione il cui tipo del valore di ritorno coincide con il nome della directory. Successivamente ad ogni codice di funzione appartenente al dataset sono state cancellate le annotazioni riguardanti il tipo del valore di ritorno in quanto questa informazione viene ripresa ed "etichettata" appositamente attraverso il nome della cartella di appartenenza.
Il dataset cosí creato é composto da 17.192 file per il tipo "str", 11.096 per "bool", 5.224 per "int", 1.663 per "float", 1.661 per "dict", 373 per "list", 114 per "tuple", 34 per "set" e 12 per "complex". Data la spropositata differenza tra il numero di funzioni trovate che utilizzano "str", "int" e "bool", é possibile anche dedurre che gli altri tipi siano largamente molto mento utilizzati nel mondo reale e di conseguenza piú difficili da analizzare.
Lavori di ricerca futuri potranno quindi comprendere questo dataset per l'analisi del codice delle funzioni basandosi sui loro valori di ritorno.

# Capitolo 5

# Conclusioni

Un numeroso insieme di lavori di ricerca ha proposto algoritmi di ricerca per la type inference di Python. Tali algoritmi sono a tutti gli effetti approcci statici che utilizzano un insieme di regole e vincoli predefiniti. Come menzionato precedentemente, gli algoritmi di type inference statici risultano spesso imprecisi per colpa della natura dinamica di Python e dall'approssimazione del comportamento dei programmi relativo ad analisi statiche.
Nel 2018 Hellendoorn ha proposto DeepTyper [10], una delle prime tecniche di Machine Learning per la type inference che basa il suo training sull'utilizzo di codice TypeScript. Questo modello é in grado di predire annotazioni per un dato codice sorgente considerando un ampio contesto. Il problema di tale modello é evidenziato nelle inconsistenze delle predizioni per quanto riguarda diverse occorrenze della stessa variabile.
Pradel, successivamente, ha proposto TypeWriter [15] che é in grado di risolvere il problema dell'inferenza di tipo per Python. TypeWriter é composto da una rete neurale che considera sia il contesto del codice che informazioni inerenti il linguaggio naturale nel codice sorgente. Inoltre tale modello valida i suoi risultati impiegando strategie di ricerca combinatoriali ed un type checker esterno.
LAMBDANET é stato introdotto da Wei [16] per migliorare il problema legato a DeepTyper: é una rete neurale per la type inference di TypeScript. La sua principale idea é quella di creare un grafo delle dipendenze dei tipi che lega le variabili a cui assegnare le annotazioni a vincoli logici e suggerimenti contestuali come assegnamenti e nomi degli identificatori. Rispetto a DeepTyper, il modello proposto da Wei dimostra la sua superioritá per ció che concerne la accuracy, mentre presenta un difetto: é in grado di predire solo i 100 tipi piú comuni all'interno di un dataset.
Dato che i vincoli naturali come identificatori e commenti presentano una incerta forma di informazione, Pandi [19] ha proposto OptTyper che riguar-

| Approach | Size of type vocabulary | ML model | Type hints | | |
|---|---|---|---|---|---|
| | | | Contexual | Natural | Logical |
| Type4Py | Unlimited | HNN (2x RNNs) | ✓ | ✓ | ✗ |
| JSNice [23] | 10+ | CRFs | ✓ | ✓ | ✗ |
| Xu et al. [24] | - | PGM | ✗ | ✓ | ✓ |
| DeepTyper [12] | 10K+ | biRNN | ✓ | ✓ | ✗ |
| NL2Type [13] | 1K | LSTM | ✗ | ✓ | ✗ |
| TypeWriter [14] | 1K | HNN (3x RNNs) | ✓ | ✓ | ✗ |
| LAMBDANET [25] | 100 | GNN | ✓ | ✓ | ✓ |
| OptTyper [26] | 100 | LSTM | ✗ | ✓ | ✓ |
| Typilus [15] | Unlimited | GNN | ✓ | ✓ | ✗ |

Figura 5.1: Confronto tra le principali tecniche illustrate

da l'inferenza di tipo per TypeScript. L'idea centrale di questo approccio é quella di estrarre informazioni deterministiche o vincoli logici a partire dal type system combinandoli con vincoli logici in un singolo problema di ottimizzazione. Questo permette di fare predizioni di tipo corrette senza violare le regole dei tipi del linguaggio. OptTyper é stato mostrato che sia migliore in questo ambito rispetto sia a DeepTyper che a LAMBDANET.
Tutti i metodi sopra elencati utilizzano un vocabolario di al massimo 1000 tipi. Per ovvi motivi viene ridotta l'abilitá per l'inferenza di tipi rari e definiti dall'utente. Per questo motivo é stato introdotto Typilus, un modello che integra informazioni di tipo contestuale come identificatori, strutture sintattiche e il flusso dei dati per l'inferenza di annotazioni per Python. Questo modello é differente dagli ultimi elencati perché ha introdotto una tecnica capace di eliminare i limiti imposti dalla grandezza del vocabolario.
Uno degli ultimi metodi creati é Type4Py, un modello che é stato in grado di migliorare la tecnica di Typilus facendo in modo che i cluster in Type4Py fossero ben separati e piú compatti rispetto a quelli in Typilus.
Per quanto riguarda un problema abbastanza esteso all'interno del Machine Learning, e piú nello specifico per quanto riguarda le tecniche utilizzate per la type inference, risulta essere l'assenza di un dataset per l'addestramento delle reti neurali. In questa tesi viene presentato un dataset creato a partire dall'idea di Mir. Egli ipotizza come all'interno di progetti Python che includano dipendenze alle librerie di type checker come mypy siano piú probabilmente presenti file sorgenti che incorporino annotazioni riguardanti i tipi. Il dataset creato quindi é composto da progetti principalmente scaricati da GitHub scritti in Python. Da tale dataset é stato poi possibile crearne uno nuovo che classificasse le funzioni contenute nei progetti in base al loro valore di ritorno.

Idee che possano contribuire allo Stato dell'Arte potranno quindi riguardare l'analisi del codice mediante l'uso del dataset descritto o la creazione di un nuovo insieme di progetti che comprenda codici scritti in un altro linguaggio dinamico come JavaScript.

# Bibliografia

[1] Z. Gao, C. Bird, and E. T. Barr. "to type or not to type: quantifying detectable bugs in javascript". *Proceedings of the 39th International Conference on Software Engineering*, pages 758–769, 2017.

[2] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. "an empirical study on the impact of static typing on software maintainability". *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[3] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. "a large scale study of programming languages and code quality in github". *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, page 155–165, 2014.

[4] B. Dagenais and M. P. Robillard. "recovering traceability links between an api and its learning resources". *Proceedings of the 34th International Conference on Software Engineering*, page 47–57, 2012.

[5] P. C. Rigby and M. P. Robillard. "discovering essential code elements in informal documentation". *Proceedings of the 35th International Conference on Software Engineering*, page 832–841, 2013.

[6] S. Subramanian, L. Inozemtseva, and R. Holmes. "live api documentation". *Proceedings of the 36th International Conference on Software Engineering*, page 643–652, 2014.

[7] H. Phan, H. Nguyen, N. Tran, L. Truong, A. Nguyen, and T. Nguyen. "statistical learning of api fully qualified names in code snippets of online forums". *Proceedings of the 40th International Conference on Software Engineering*, page 632–642, 2018.

[8] C. M. K. Saifullah, M. Asaduzzaman, and C. K. Roy. "learning from examples to find fully qualified names of api elements in code snippets".

*Proceedings of the 34th International Conference on Automated Software Engineering*, 2019.

[9] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery.

[11] http://mypy-lang.org/.

[12] https://www.typescriptlang.org/.

[13] https://flow.org.

[14] Casper Boone, Niels de Bruin, Arjan Langerak, and Fabian Stelmach. Dltpy: Deep learning type inference of python function signatures using natural language context, 2019.

[15] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 209–220, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks, 2020.

[17] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Deep similarity learning-based type inference for python, 2021.

[18] Yun Peng, Zongjie Li, Cuiyun Gao, Bowei Gao, David Lo, and Michael Lyu. Hityper: A hybrid static type inference framework with neural prediction, 2021.

[19] Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. Opttyper: Probabilistic type inference by optimising logical and natural constraints, 2021.

[20] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. Pyinfer: Deep learning semantic type inference for python variables, 2021.

[21] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 607–618, New York, NY, USA, 2016. Association for Computing Machinery.