Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Informatica

Machine Learning Based Programming Language Identification

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Matteo Conti

Sessione I
Anno Accademico 2019-2020

So much depends
upon
a red wheel
barrow
glazed with rain
water
beside the white
chickens.

- William Carlos Williams

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

L'avvento dell'era digitale ha contribuito allo sviluppo di nuovi settori tecnologici, i quali, per diretta conseguenza, hanno portato alla richiesta di nuove figure professionali capaci di assumere un ruolo chiave nel processo d'innovazione tecnologica.

L'aumento di questa richiesta ha interessato particolarmente il settore dello sviluppo del software, a seguito della nascita di nuovi linguaggi di programmazione e nuovi campi a cui applicarlo. La crescita esponenziale del numero degli stessi linguaggi, che rappresentano il 'cuore' di questo settore, rispecchia senza dubbio l'impossibilità per chiunque di operare in qualsiasi di questi disinteressandosi di questo aspetto.

L'aumento di figure professionali in grado di 'creare' *software* ha contribuito alla crescita delle risorse online all'interno delle quali è possibile trovare frammenti di codice sorgente. Infatti, navigando online, è possibile trovare guide/documentazioni tecniche relative ad uno specifico linguaggio, blog contenenti consigli e aggiornamenti sugli ultimi trend, forum dove poter confrontarsi liberamente su aspetti tecnici altamente settorializzati, piattaforme di e-learning per imparare e/o approfondire aspetti rilevanti e molto altro (chiaramente la quantità di risorse è proporzionale alla popolarità di uno specifico linguaggio).

A questo bisogna aggiungere il numero di progetti pubblicati sulle principali piattaforme online di *version-control* basate sulla tecnologia open-source Git (a Gennaio 2020, la piattaforma Github ha dichiarato di avere oltre 40 milioni di utenti e più di 100 milioni di 'repository' tra le quali almeno 28 milioni pubbliche) [3].

La componente principale di cui è composto un software è il *codice sorgente*, che può

essere descritto come un archivio di uno o più file testuali contenti una serie d'istruzioni scritte in uno o più linguaggi di programmazione (a basso/alto livello).

Nonostante molti di questi linguaggi vengano utilizzati in diversi settori tecnologici, spesso accade che due o più di questi condividano una struttura sintattica e semantica molto simile. Questo chiaramente deriva dal fatto che, ad ogni linguaggio, il più delle volte è possibile attribuire un 'predecessore', ovvero un linguaggio usato come 'base' per costruire quest'ultimo. Chiaramente questo aspetto può generare confusione nell'identificazione di questo all'interno di un frammento di codice, soprattutto se consideriamo l'eventualità che non sia specificata nemmeno l'estensione dello stesso file (la quale contribuirebbe in modo significativo alla semplificazione di questo problema). Infatti, ad oggi, la maggior parte del codice disponibile online contiene informazioni relative al linguaggio di programmazione specificate manualmente.

All'interno di questo elaborato ci concentreremo nel dimostrare che l'identificazione del linguaggio di programmazione di un file 'generico' di codice sorgente può essere effettuata in modo automatico utilizzando algoritmi di *Machine Learning* e non usando nessun tipo di assunzione 'a priori' sull'estensione o informazioni particolari che non riguardino il contenuto del file.

Questo progetto segue la linea dettata da alcune ricerche precedenti basate sullo stesso approccio, confrontando tecniche di estrazione delle *features* differenti e algoritmi di classificazione con caratteristiche molto diverse, cercando di ottimizzare la fase di estrazione delle *features* in base al modello considerato.

Il codice sorgente di questo progetto è disponibile sotto licenza MIT al seguente link: `https://github.com/contimatteo/Programming-Language-Identification`.

Capitolo 1

Approccio

Lo scopo di questo capitolo è fornire una panoramica dei concetti chiave e dell'approccio attorno al quale si sviluppa questo elaborato di tesi.

Nel capitolo 2 verranno descritti tutti i passaggi relativi al pre-processamento ('trasformazione') del dataset estratto dall' archivio usato successivamente per la estrazione delle *features*. Nei capitoli 3, 4 e 5 descriveremo i 3 classificatori analizzati, introducendo gli aspetti teorici e i processi aggiuntivi di selezione delle features. Nel capitolo 6 mostreremo i report dettagliati riguardo all'accuratezza ottenuta, esponendo le metriche di classificazione per ognuno dei vari modelli scelti.

1.1 Obiettivi

Il mondo del web ospita un numero elevatissimo di archivi di codice sorgente e le risorse a disposizione crescono proporzionalmente con l'avanzare del tempo. Tuttavia, a causa delle grandi dimensioni di molti archivi recenti e non, è diventato oramai troppo costosa la categorizzazione manuale del codice sorgente [4].

Prima di proseguire con le specifiche del progetto, dobbiamo mettere in dubbio l'utilità di questa ricerca ponendoci una semplice domanda: "*è davvero necessario classificare in modo automatico il codice sorgente?*". La migliore argomentazione a sostegno di questo quesito è sicuramente la lista dei casi d'uso a cui sarebbe possibile applicare i risultati ottenuti da questo progetto: evidenziazione della sintassi all'interno di editor testuali/online, indicizzazione di repository di

codice sorgente, stima delle tendenze nella popolarità di un singolo linguaggio e suggerimenti sull'etichettatura degli snippet su piattaforme come StackOverflow [5].

Di fatto, la realizzazione di queste funzionalità sarebbe notevolmente semplificata con l'aiuto di un classificatore automatico in grado di identificare il linguaggio di programmazione di uno snippet di codice.

L'obiettivo della **classificazione del testo** è la categorizzazione dei documenti (nel nostro caso, file di codice sorgente) in un numero fissato di classi (nel nostro caso, linguaggi di programmazione) [5]. Nonostante le differenze di struttura e sintassi, tutti i linguaggi di programmazione, proprio come il linguaggio naturale, hanno caratteristiche e parole chiave (specifiche) che possono essere facilmente identificate.

Da questo punto di vista, questo progetto dimostra diverse affinità con le tecniche di classificazione usate principalmente in problemi riguardanti il riconoscimento del testo.

1.2 Aspetti Rilevanti

L'estrazione delle features è il primo passo per la costruzione di un classificatore ed è sicuramente quello più importante. Per la costruzione e l'identificazione di queste all'interno del nostro problema sono fondamentali quelli che in seguito definiremo come **tokens**, ovvero stringhe formate da una e/o più caratteri/parole all'interno di un singolo file di codice.

Inoltre, ricordiamo come all'interno di uno snippet di codice non siano presenti sempre e solo frasi e/o parole appartenenti ad un preciso linguaggio: spesso è possibile trovare diversi 'elementi' corrispondenti al linguaggio naturale, come ad esempio il testo usato nei commenti oppure informazioni relative all'autore o alla licenza del file. Sottovalutare questo aspetto corrisponde sicuramente ad allontanarsi dal trovare una soluzione 'valida' al problema trattato in questo progetto.

Inoltre, è evidente come il riconoscimento di alcune parole chiave di uno specifico linguaggio A, all'interno dei commenti di un file scritto nel linguaggio B, rischi di condizionare in modo eccessivo la classificazione.

Per questi motivi la fase di pre-processamento di ogni file (ovvero la fase in cui cercheremo di eliminare i conflitti sopra descritti) diventa il punto cruciale per effettuare un'estrazione corretta

delle features.

Al fine di confrontare l'efficacia dei nostri classificatori (uno per ogni modello), misuriamo le percentuali di *veri-positivi* e *falsi-positivi* per ogni linguaggio, oltre all'accuratezza complessiva.

1.3 Dataset

Il dataset di riferimento usato per questo progetto si chiama *Rosetta-Code*, ovvero un archivio online contenente frammenti di codice sorgente creati al fine di presentare le soluzioni per un insieme di problemi fissati nel maggior numero possibile di linguaggi di programmazione. L'obiettivo di questo dataset è dimostrare in che modo linguaggi diversi presentino somiglianze e/o differenze grazie al confronto delle rispettive soluzioni per lo stesso problema. Questo archivio contiene attualmente 1,058 problemi e un numero elevato di soluzioni scritte in 789 linguaggi di programmazione diversi (per ovvi motivi, non tutti i problemi contengono una soluzione per ogni linguaggio) [6].

Il dataset è stato scaricato da una repository di Github [7] nata con lo scopo innanzitutto di replicare il dataset già presente sul sito ufficiale, che a semplificare l'accesso agli esempi. Chiaramente, per i motivi appena elencati, il numero di esempi presenti per ogni linguaggio non corrisponde precisamente a quello del sito originale, ma ne fornisce un numero sufficiente (in alcuni casi anche superiore).

Per ottenere risultati consistenti non considereremo tutti i linguaggi presenti nel dataset, ma solo quelli contenenti almeno 400 snippet di codice. Ecco la lista completa con indicato il numero di esempi per ciascuno:

linguaggio	**esempi**	**linguaggio**	**esempi**	**linguaggio**	**esempi**
ALGOL-68	588	Go	1130	Phix	885
AWK	473	Haskell	1427	PicoLisp	851
Ada	1045	Icon	598	PowerShell	816
AutoHotkey	669	J	1879	PureBasic	590
BBC-BASIC	534	Java	1051	Python	1677
C	1139	JavaScript	1186	R	664
C++	926	Jq	1042	REXX	1148
Clojure	768	Julia	990	Racket	987

continua nella prossima pagina

Tabella 1.1 – continuo della pagina precedente

linguaggio	**esempi**	**linguaggio**	**esempi**	**linguaggio**	**esempi**
Common-Lisp	1014	Kotlin	769	Ring	500
D	1009	Lua	622	Ruby	1125
Elixir	536	Mathematica	949	Rust	580
Erlang	528	Nim	660	Scala	931
Factor	637	OCaml	862	Sidef	787
Forth	603	PARI-GP	614	Swift	505
Fortran	771	Perl	1223	Tcl	1156
FreeBASIC	431	Perl-6	1223	Zk	1049
	11671		**16225**		**14251**

Tabella 1.1: Numero di linguaggio ed esempi considerati.

Quindi, in totale, procederemo al riconoscimento di 48 linguaggi diversi usando un dataset formato da 42147 esempi.

1.4 Ricerche Correlate

Nel corso degli anni, diverse ricerche sono state condotte utilizzando tecniche di apprendimento per la classificazione del testo: 'Neural Networks', 'Decision Treee', 'Naive Bayes', 'Support Vector Machines' e 'Nearest Neighbor Classifiers' [8]. All'interno di questo elaborato presentiamo le specifiche di un progetto basato su alcuni tra questi algoritmi di classificazione, riproponendo tecniche di estrazione e selezione delle features già presenti in alcune di queste ricerche:

autori	**metodo**	**features**	classi	esempi	% accur.
Ugurel et. al. [5]	Support Vector Machine	Sequenze di 1-gram alfanumerici separati da caratteri non alfanumerici.	10	300	≈ 89%
Khasnabish et. al. [9]	Multinomial Naive Bayes	Frequenza di ciascuna keyword.	10	≈ 2K	≈ 93%

continua nella prossima pagina

Tabella 1.2 – continuo della pagina precedente

autori	**metodo**	**features**	classi	esempi	% accur.
Kennedy van Dam et. al. [10]	Classificatore Kneser-Ney modificato	1-gram, 2-gram e 3-gram separati da spazi.	20	≈ 4K	≈ 96%
Shaul Zevim and Catherine Holzem [11]	Classificatore di Massima Entropia	1-gram, 2-gram e 3-gram di 'tokens' lessicali.	29	≈ 147K	≈ 99%
Shlok Gilda [8]	Rete Neurale Convoluzionale	'tokens' lessicali rappresentati sotto forma di matrice (word-embeddings).	60	≈ 118K	≈ 97%
Anonymous Authors [4]	Rete Neurale	1-gram, 2-gram e 3-gram separati da spazi, punteggiatura e caratteri speciali.	121	≈ 2M	≈ 85 %

Tabella 1.2: Lista delle principali ricerche correlate.

Analizzando le ricerche della Tabella 1.2, abbiamo riscontrato come la maggior parte di queste, nonostante l'utilizzo di classificatori e fasi di pre-processamento diverse, presentino diversi elementi in comune all'interno della fase di selezione delle features.

Confrontando queste ricerche con il progetto illustrato in questo elaborato di tesi possiamo evidenziare che:

- queste si basano su dataset prelevati da repository di codice (es: Github) creando quindi un dataset molto diverso rispetto al nostro (come spiegato nella sezione 1.3).
- procedono all'estrazione di tokens formati n-gram, mentre noi lavoriamo esclusivamente con tokens 1-gram.
- la logica di eliminazione o sostituzione dei tokens nella fase di pre-processamento è basata unicamente sulla frequenza con cui compare il token (all'interno di una specifica classe

o in tutto il dataset) e, al contrario invece, oltre a questa logica, noi procediamo anche rimuovendo 1-gram alfanumerici contenenti un solo carattere o esclusivamente numerici.

- non prevedono logiche di eliminazione dei caratteri speciali, mentre in questo progetto vengono proposte due 'versioni': una mantenendo i caratteri speciali e una eliminandoli.

L'obiettivo principale di questo progetto, oltre a quello di analizzare un problema specifico, è quello di confrontare i risultati ottenuti nelle ricerche precedenti tenendo in considerazione due aspetti importanti:

- evidenziare come influisce la presenza/assenza dei caratteri speciali rispetto all'accuratezza finale.
- confrontare l'accuratezza finale dei vari classificatori considerando un dataset 'diverso'.
- confrontare quanto le differenze nella fase di pre-processamento influiscano sulla 'accuratezza' finale dei classificatori utilizzati.

Capitolo 2

Pre-Processamento

In questo capitolo presenteremo nel dettaglio il pre-processamento applicato sul dataset descritto nella sezione 1.3, processamento che viene applicato a tutti i modelli usati.

L'obiettivo di questa fase è cercare di rimuovere, a monte del modello usato, tutte le combinazioni di caratteri non significative, ovvero che non dipendono da uno specifico linguaggio (es: commenti, numeri, ...).

2.1 Preparazione e Bilanciamento

La prima fase prevede la 'ristrutturazione' del nostro dataset, ovvero provvediamo a clonare gli snippet di codice originali cercando di ottenere una struttura più performante, in modo da poter confrontare il risultato finale del nostro pre-processamento e allo stesso tempo analizzare le corrispondenze tra sorgenti di linguaggi diversi.

Per ogni esempio consideriamo due sorgenti: il primo è il sorgente **originale**, ovvero uno snippet che contiene esattamente il codice presente nel dataset originale, mentre il secondo è quello **parsato**, ovvero uno snippet che conterrà il risultato delle operazioni definite nelle prossime sezioni.

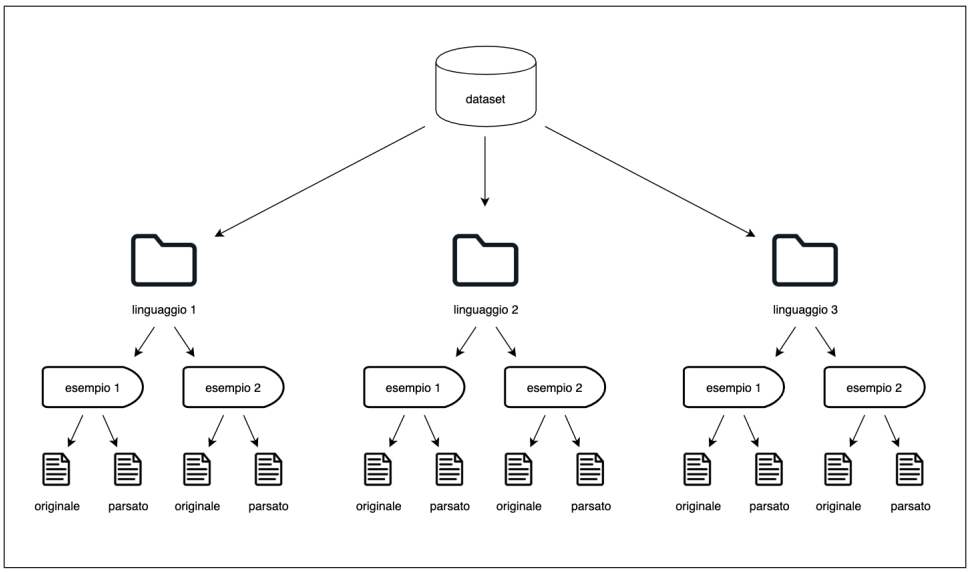

Figura 2.1: Struttura del dataset modificata.

Una volta ottenuto la seguente struttura, procediamo effettuando un'operazione chiamata **bilanciamento** del dataset. Nella maggior parte dei casi l'obiettivo di questa procedura è ottenere un dataset formato n classi dove ognuna contenga m esempi.

Per eseguire questo processo esistono diverse tecniche divise principalmente in due categorie: **sovra-campionamento** e **sotto-campionamento**.

Nel primo caso, vengono generate nuove 'istanze' affinché le classi con un minor numero di esempi raggiungano la stessa 'cardinalità' di quelle con un numero più alto di questi. La generazione delle nuove 'istanze' può avvenire attraverso la clonazione di esempi esistenti, oppure, nel caso sia possibile, attraverso la 'creazione' di esempi basandosi su proprietà statiche presenti in quella determinata classe. Mentre nel secondo caso, vengono eliminati tutti gli esempi (in modo causale) sopra una certa soglia fissata [4].

Nel nostro caso, essendo lo squilibrio tra alcune classi molto significativo, usando una tecnica di sovra-campionamento avremmo dovuto usare gli stessi esempi (o le stesse informazioni estratte da pochi di questi) troppe volte, aumentando così il rischio di overfitting. Inoltre, è evidente che usando il secondo approccio saremmo limitati dal numero di istanze della classe meno popolata.

Considerando però il problema affrontato, e ammettendo di aver a disposizione un dataset

'sensato' (ovvero con pochi duplicati), è chiaro come un elevata quantità di file sorgenti sia fondamentale per cercare di catturare in modo preciso le 'proprietà identificative' di ciascun linguaggio.

Per i motivi appena descritti abbiamo applicato una tecnica di sotto-campionamento: per ogni linguaggio, abbiamo selezionato in modo casuale 400 esempi da inserire **training-set**, mentre quelli restanti sono stati inseriti nel **testing-set**.

2.2 Tokenizzazione e Processamento

La 'tokenizzazione' di un file testuale corrisponde alla divisione del file in **tokens** ottenuti dalla concatenazione dei caratteri all'interno dello stesso file. Questa procedura viene applicata spesso a problemi nell'ambito del processamento del linguaggio naturale, in cui la separazione delle parole dalla punteggiatura è fondamentale.

Nel nostro caso, eseguiremo questa procedura due volte, la prima mantenendo i caratteri speciali, mentre la seconda eliminandoli.

A prescindere della presenza o meno di questi caratteri, consideriamo come **tokens** tutte le sequenze di caratteri alfanumerici separate da spazi, caratteri speciali e punteggiatura. Considerando come esempio il seguente snippet:

```
for p in range(0, N+1) :
```

otteniamo i seguenti **tokens**:

```
for, p, in, range, (, 0, N , +, 1, ), :
```

Commenti procediamo rimuovendo tutte le sequenze di **tokens** alfanumerici all'interno della stessa linea, escludendo a priori il primo token (ovvero simbolo usato per definire il commento), che soddisfino la seguente espressione regolare '`^[a-zA-Z ]*$`' (commenti su una riga). Essendo quest'espressione molto generica però, procederemo all'eliminazione solo nel caso in cui questa non contenga una delle parole chiave riservate definite a priori e contenga un numero di tokens superiore a 3. Considerando come esempio il seguente snippet:

```
# 9 billion names of god the integer
```

e la seguente lista di parole riservate:

```
break, catch, class, echo, else, end, defmodule, package, for, func, function,
  global, if, import, procedure, return, static, struct, try, var, with, while
```

questa riga risulterà positiva al confronto con l'espressione regolare sopra descritta e di conseguenza verrà eliminata.

Lettere procediamo sostituendo tutti **tokens** alfanumerici di lunghezza 1 con una sequenza prefissata. Considerando come esempio il seguente snippet:

```
diffs.extend([(2*k - 1, s), (k, s)])
```

otteniamo una nuova linea così generata:

```
diffs . extend ([( 2 * __a__ - 1 , __a__ ), ( __a__ , __a__ )])
```

Numeri procediamo sostituendo tutti **tokens** numerici con una sequenza prefissata. Considerando come esempio il seguente snippet:

```
while k * (3*k-1) < 2*N:
```

otteniamo una nuova linea così generata:

```
while k * ( __n__ * k - __n__ ) < __n__ * N :
```

Quindi, considerando come esempio il seguente file di codice sorgente scritto nel linguaggio Python:

```
# 9 billion names of god the integer
def partitions(N):
    diffs,k,s = [],1,1
    while k * (3*k-1) < 2*N:
        diffs.extend([(2*k - 1, s), (k, s)])
        k,s = k+1,-s

    out = [1] + [0]*N
    for p in range(0, N+1):
```

```
        x = out[p]
        for (o,s) in diffs:
            p += o
            if p > N: break
            out[p] += x*s

    return out

p = partitions(12345)
for x in [23,123,1234,12345]: print x, p[x]
```

Snippet 2.1: esempio di file sorgente

otterremo alla fine del processo sopra descritto il seguente file parsato (nel caso in cui i caratteri speciali non vengano eliminati):

```
def partitions ( __a__ ):
    diffs , __a__ , __a__ = [], __n__ , __n__
    while __a__ * ( __n__ * __a__ - __n__ ) < __n__ * __a__ :
        diffs . extend ([( __n__ * __a__ - __n__ , __a__ ), ( __a__ , __a__ )])
        __a__ , __a__ = __a__ + __n__ , - __a__

    out = [ __n__ ] + [ __n__ ]* __a__
    for __a__ in range ( __n__ , __a__ + __n__ ):
        __a__ = out [ __a__ ]
        for ( __a__ , __a__ ) in diffs :
            __a__ += __a__
            if __a__ > __a__ : break
            out [ __a__ ] += __a__ * __a__

    return out

__a__ = partitions ( __n__ )
for __a__ in [ __n__ , __n__ , __n__ , __n__ ]: print __a__ , __a__ [ __a__ ]
```

Snippet 2.2: esempio del risultato della fase di pre-processamento su un file sorgente

Concludiamo la fase di pre-processamento eliminando tutti i **tokens** "`__a__`" e "`__n__`", oltre che ai caratteri "\t" e "\n".

2.3 Generazione Vocabolario

Considerando la definizione del concetto di **token**, all'interno del nostro problema, esplicitata nella fase di processamento descritta nella sezione precedente, procediamo con la fase di costruzione della struttura dati portante del seguente processo di selezione delle features: il **vocabolario**.

Nell'ambito dei problemi legati al processamento del linguaggio naturale, un passaggio fondamentale, comune a tutti questi, è il processo di rappresentazione numerica dei tokens testuali. All'interno di questi problemi, come nel nostro, i dati di input sono rappresentati in formato testuale, e dopo una fase iniziale (pre-processamento) si prosegue, attraverso l'uso di diverse tecniche (es: word-embedding, TF-IDF, ...), al processo di rappresentazione del formato (testuale) in input in formato numerico (vettori). Nonostante esistano varie procedure, il risultato finale di ciascuna di esse consiste nell'assegnazione di valori interi o reali a ciascun **token** (i quali sono definiti in modo arbitrario in base al problema affrontato).

In molti di questi problemi però l'univocità di un token risulta fondamentale: se consideriamo come esempio il problema affrontato in questo elaborato, risulta evidente come sia necessario identificare ogni token in modo **univoco**, in modo tale da catturare la peculiarità di alcune *keywords* all'interno di un singolo linguaggio.

Per questo motivo, si procede alla costruzione di una struttura chiamata **vocabolario**: questo è formato da una mappa (dizionario) di coppie chiave valore, dove ciascuna chiave corrisponde a un **token** e un valore intero/reale generato (casualmente o no) in modo tale che questo sia univoco all'interno di tutta la struttura.

A seguito del primo test, in cui abbiamo escluso i caratteri speciali, il vocabolario generato dopo la fase di pre-processamento conteneva $\approx$ 13K tokens, mentre nel secondo test, dove non abbiamo escluso nessuno di questi caratteri, il vocabolario conteneva $\approx$ 18K tokens.

Capitolo 3

Multinomial Naive Bayes

In questo capitolo, per la procedura di estrazione delle features, non utilizziamo nessun riferimento a ricerche precedenti, nonostante l'uso di un classificatore molto utilizzato all'interno di problemi relativi al riconoscimento del testo e utilizzato in un ricerca prima citata [9]).

I metodi chiamati *Naive Bayes* sono un insieme di algoritmi supervisionati (*supervised learning*) basati sull'applicazione del teorema di Bayes con l'assunzione *naive* dell'indipendenza condizionale tra tutte le coppie di *tokens* fissato il valore di una specifica classe [12].

Il classificatore *Multinomial Naive Bayes* implementa il teorema di Bayes per i dati distribuiti multinomialmente ed è una delle due varianti classiche utilizzate nella classificazione del testo. La distribuzione è parametrizzata da vettori $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ per ogni classe y, dove n è il numero di caratteristiche (nella classificazione del testo, ovvero la dimensione del vocabolario) ed θ_{yi} è la probabilità $P(x_i|y)$ di un *token* i di apparire in un file sorgente appartenente alla classe y.

Il disaccoppiamento delle distribuzioni delle caratteristiche (condizionali) di ogni classe implica che ogni distribuzione può essere stimata indipendentemente in forma monodimensionale [12].

3.1 Background

Teorema 1 (Teorema di Bayes)**.** *Considerando un insieme di classi* $C = (c_1, \ldots, c_n)$ *che partizionano lo spazio degli eventi* Ω $(c_i \cap c_j = \varnothing \ , \forall i \neq j \wedge \cup_{i=1}^{n} c_i = \Omega)$ *si trova la seguente*

espressione per la probabilità condizionata:

$$P(c_i|d) = \frac{P(d|c_i)P(c_i)}{P(d)} = \frac{P(d|c_i)P(c_i)}{\sum_{j=1}^{n} P(d|c_j)P(c_j)}$$

Il nostro obiettivo infatti sarà determinare per ogni evento la C_{MAP} (MAP è l'abbreviazione di "massima a posteriori", ovvero la classe più probabile) definita come:

$$C_{MAP} = \underset{c \in C}{\operatorname{argmax}} P(c|d)$$

dove per il teorema di Bayes

$$C_{MAP} = \underset{c \in C}{\operatorname{argmax}} \frac{P(d|c)P(c)}{P(d)} = \underset{c \in C}{\operatorname{argmax}} P(d|c)P(c)$$

Applicando questo teorema al nostro problema iniziale, possiamo considerare ogni linguaggio come una classe c_i e ogni file di codice sorgente come un evento d.

3.2 Assunzioni d'Indipendenza

Abbiamo già accennato alla possibilità di rappresentare un singolo file di codice sorgente come la concatenazione di una serie di *tokens* diversi tra loro. Usando questa prospettiva, potremmo sfruttare il teorema definito nella sezione precedente, considerando però ogni evento d come un vettore di componenti indipendenti $(x_1, \ldots, x_n)$, dove ciascuna componente corrisponde ad un token nel i-esimo file di codice sorgente. In questo modo otteniamo che:

$$P(d|c) = P(x_1, \ldots, x_n|c)$$

Rispetto all'uguaglianza sopra definita, elenchiamo una serie di assunzioni riguardo il concetto di indipendenza:

- La posizione di ciascun token x_i non è rilevante
- la probabilità $P(x_i|c_j)$ di ogni feature x_i è indipendente fissata una classe c_j

Da queste assunzioni deriviamo:

$$C_{MAP} = \underset{c \in C}{\operatorname{argmax}} P(x_1, \ldots, x_n|c)P(c)$$

3.3 Estrazione delle Features

Prima di procedere all'estrazione delle features, ricorsivamente, per ogni linguaggio, accediamo al contenuto di ogni file sorgente presente nel training-set, rimuovendo tutti i tokens che non compaiono all'interno di almeno 4 esempi (10% su 400) della classe a cui appartiene.

A questo punto possiamo procedere creando, per ogni esempio, la lista delle ***features***: preso in input un file, sostituiamo ogni token del nostro vocabolario (descritto nella sezione sezione 2.3) con il suo numero di occorrenze dello stesso token all'interno di questo file. In questo modo ogni snippet del dataset conterrà lo stesso numero di features, il quale corrisponderà al numero totale di tokens all'interno del nostro vocabolario.

3.4 Esempio

Consideriamo un file di codice sorgente scritto nel linguaggio "Python":

```
def partitions __a__
    partitions __a__ append __n__
    for __a__ in xrange __n__ __a__ __n__
        __a__ __a__ __a__ __n__ __a__ __n__ __n__
        if __a__ __n__
        if __a__ __n__
            partitions __a__ __a__ partitions __a__ __a__
        else
            partitions __a__ __a__ partitions __a__ __a__
        __a__ __a__
        if __a__ __n__

            if __a__ __n__
                partitions __a__ __a__ partitions __a__ __a__
            else
                partitions __a__ __a__ partitions __a__ __a__
    return partitions __a__ __n__

partitions __a__ __n__

def main
    ns set __n__ __n__ __n__ __n__
```

```
    max ns max ns
    for __a__ in xrange __n__ max ns __n__
        if __a__ max ns
            __a__ partitions __a__
        if __a__ in ns
            print 6d __a__ __a__ __a__

main
```

Snippet 3.1: Naive Bayes: esempio di un file sorgente.

e consideriamo un **vocabolario** ipotetico:

```
{
    'def': 1,           'return': 2,        'partitions': 3,
    'str': 4,           'append': 5,        'raise': 6,
    'for': 7,           'of': 8,            'in': 9,
    'xrange': 10,       'main': 11,         'include': 12,
    'ns': 13,           'max': 14,          'from': 15,
    'print': 16,        'if': 17,           'dict': 18,
    'normalize': 19,    'global': 20
}
```

Snippet 3.2: Naive Bayes: esempio di vocabolario.

e la lista di ***features*** generate sarà:

```
[ 2, 1, 13, 0, 1, 0, 2, 0, 3, 2, 2, 0, 6, 4, 0, 1, 6, 0, 0, 0 ]
```

Snippet 3.3: Naive Bayes: esempio di features.

Capitolo 4

Support Vector Machine

In questo capitolo descriviamo l'applicazione di un classificatore e di una procedura di estrazione delle features già utilizzati in una ricerca precedente [5], in modo tale da poter confrontare i risultati ottenuti in questa con quelli ottenuti sul nostro dataset.

Support Vector Machines sono una classe di algoritmi supervisionati (*supervised learning*) principalmente usati nell'ambito dei problemi di classificazione e regressione.

Questa tipologia di algoritmi è facilmente applicabile ai problemi di *riconoscimento del testo* e, se applicata a questi, garantisce performance migliori rispetto ad altri metodi [13].

La sua capacità di gestire vettori di *features* di elevate dimensioni per ridurre i problemi derivati da un eccesso di *overfitting*, rende questi algoritmi un'ottima tecnica per le attività di classificazione del testo [14].

4.1 Background

L'algoritmo Support Vector Machine costruisce un iperpiano (o un insieme di iperpiani) lineare in uno spazio di dimensione elevata o infinita. Intuitivamente, una buona separazione è ottenuta dall'iperpiano che forma la distanza maggiore tra i punti di *training* più vicini di qualsiasi classe (questo iperpiano viene chiamato *margine funzionale*), poiché, in generale, maggiore è il margine, minore è l'errore di generalizzazione del classificatore [1].

Se dati del problema non sono linearmente separabili nello spazio originale, allora lo saranno sicuramente in uno spazio di dimensione maggiore.

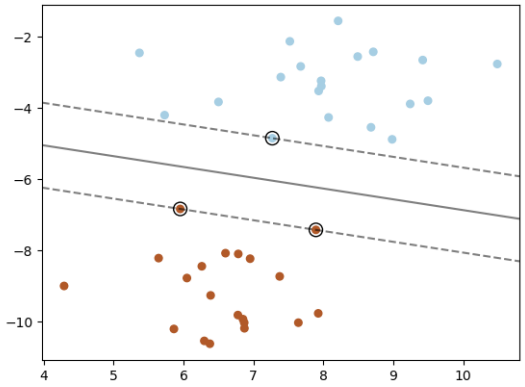

Figura 4.1: Esempio SVM [1]

La figura seguente mostra la funzione decisionale per un problema linearmente separabile (contenente due classi). Possiamo notare come le linee tratteggiate corrispondano al *'confine'* di ogni classe (o iperspazio), mentre la linea continua corrisponde al *margine funzionale*.

Uno dei punti chiave di questo algoritmo è che alcuni input sono più importanti di altri, e, di conseguenza, concentrare l'attenzione su questi porta al raggiungimento di una generalizzazione più precisa [15].

4.2 Estrazione delle Features

Per questo algoritmo, basandoci su una delle ricerche precedenti [5], abbiamo deciso di procedere all'estrazione solo di un sotto-insieme, dei *tokens* di ogni file di codice sorgente appositamente selezionato.

Per fare ciò, dobbiamo procedere a calcolare la **misurazione della perdita d'entropia attesa** di ogni token. Una selezione delle features basata su questo concetto, aumenta l'efficacia e l'efficienza attraverso la rimozione dei termini non informativi secondo la *corpus statistics* [16].

Formule per il calcolo dell'entropia [5]. Sia C l'evento che indica se un file sorgente è scritto in un determinato linguaggio (classe), sia f l'evento che indica se un file sorgente contiene un *token* fissato e Pr() la loro probabilità:
L'entropia iniziale (della distribuzione) di una classe è

$$e = -\Pr(C)\lg\Pr(C) - \Pr(\bar{C})\lg\Pr(\bar{C})$$

L'entropia di una classe quando il *token* è presente è

$$e_f = -\Pr(C|f)\lg\Pr(C|f) - \Pr(\bar{C}|f)\lg\Pr(\bar{C}|f)$$

L'entropia di una classe quando il *token* non è presente è

$$e_{\bar{f}} = -\Pr(C|\bar{f})\lg\Pr(C|\bar{f}) - \Pr(\bar{C}|\bar{f})\lg\Pr(\bar{C}|\bar{f})$$

La **perdita d'entropia attesa** per il *token* fissato è

$$el_{\text{expected}} = e - e_f\Pr(f) + e_{\bar{f}}\Pr(\bar{f})$$

Questa misurazione è sempre non negativa, e, maggiore è il suo valore, tanto più il *token* in questione è rilevante (ovvero la sua presenza/assenza influisce maggiormente sulla probabilità che un sorgente appartenga a una classe).

Selezione delle Features. Procediamo applicando le formule generiche della sezione precedente al nostro problema e ai nostri dati di input. Fissato un linguaggio (classe), i file sorgenti che appartengono a questo sono considerati come *'positivi'* mentre tutti gli altri sono considerati *'negativi'*. Calcoliamo le probabilità prima menzionate come segue:

$$\Pr(C) = \frac{\text{numero di esempi positivi}}{\text{numero di esempi}} \qquad \Pr(\bar{C}) = 1 - \Pr(C)$$

$$\Pr(f) = \frac{\text{numero di esempi con F}}{\text{numero di esempi}} \qquad \Pr(\bar{f}) = 1 - \Pr(f)$$

$$\Pr(C|f) = \frac{\text{numero di esempi positivi con F}}{\text{numero di esempi con F}} \qquad \Pr(\bar{C}|f) = 1 - \Pr(C|f)$$

$$\Pr(C|\bar{f}) = \frac{\text{numero di esempi positivi senza F}}{\text{numero di esempi senza F}} \qquad \Pr(\bar{C}|\bar{f}) = 1 - \Pr(C|\bar{f})$$

Per estrarre le features indicizziamo ciascun file sorgente e calcoliamo la *perdita d'entropia attesa* per ciascun token. Fatto ciò, i tokens vengono ordinati in base a questa metrica in ordine decrescente.

Dalla lista così ottenuta selezioniamo per ogni linguaggio (classe) i primi **15 *tokens*** (ovvero quelli più "rilevanti") presenti nella rispettiva lista.

4.3 Esempio

Seguendo il metodo definito nella sezione precedente, considerando tutti i file sorgenti (senza caratteri speciali) otteniamo la seguente lista di tokens:

linguaggio	**tokens (no caratt. speciali)**	**tokens (con caratt. speciali)**
algol-68	int, if, to, print, for, the, end, proc, do, then, of, string, in, and, is	(, „), =, ;, :, -,);, \", #, int, :=, to, [, print,
awk	for, print, begin, if, function, return, printf, awk, in, exit, length, the, while, to, of	{, (,), }, =, -, „ ;, \", ., for, if, print, begin, #,
ada	end, is, with, in, return, for, begin, if, procedure, function, use, loop, then, of, type	(, ;, :, end, .,), is, „);, with, :=, in, begin, for, procedure,
autohotkey	return, if, loop, msgbox, in, for, the, of, to, is, a_index, while, else, and, parse	„ (,), %, }, \", {, ., =, :=, -, return, ;, if, :,
bbc-basic	print, to, if, for, def, then, dim, local, and, of, the, while, next, rem, is	=, (,), „ \", print, %, to, if, :, for, -, +, def, .,
c	int, return, main, for, if, include, printf, stdio, char, void, while, the, string, to, is	(, ., ;, =, }, „), {, <, int, #, >, return,);, main,
c++	int, main, return, for, std, include, if, cout, string, iostream, end, the, void, const, while	(, }, ;, {,), <, „ =, >, int, #, ., return, main, \",
clojure	let, if, println, defn, map, def, of, for, range, the, first, and, str, count, to	(,), [, -,], \",)), ., :, if,))), let, /, „ println,
common-lisp	for, if, loop, do, to, let, defun, list, format, from, and, in, the, string, with	(,), -,)), \", :,))), ((, for, „ ., let, *, if, =,

continua nella prossima pagina

Tabella 4.1 – continuo della pagina precedente

linguaggio	**tokens (no caratt. speciali)**	**tokens (con caratt. speciali)**
d	main, import, void, std, stdio, return, if, in, int, foreach, string, length, writeln, auto, algorithm	., (, ;, }, „ {,), =, main, (), import, void,);, std, [,
elixir	end, do, io, def, enum, fn, puts, if, string, each, _, for, map, length, list	., (,), „ =, :, end, do, io, \", ->, }, def, {, [,
erlang	end, module, io, fun, export, lists, _, when, main, format, true, of, ok, length, false	(, „ :,), -, =, ->, ., [,)., /, (),)„ end, {,
factor	in, using, math, print, io, kernel, dup, if, map, sequences, each, swap, main, code, rosetta	:, ., -, (,), [,], ;, \", }, {, –, >, in, using,
forth	if, then, do, dup, loop, swap, to, and, drop, else, of, over, the, create, while	:, ;, (,), ., -, +, –, \", „ if, =, \\, >, /,
fortran	integer, if, end, do, in, write, program, then, the, to, function, is, and, of, call	(,), „ =, ., integer, -, if, ::, end, +, :, do, in, \",
freebasic	print, to, as, end, for, if, dim, sleep, then, integer, any, key, the, string, next	(, =,), \", „ print, ., ', to, as, end, -, for, if, :,
go	main, import, func, if, for, package, println, fmt, int, return, string, var, to, len, _	(,), ., {, }, „ \", (), =, main, import, :=, func, if, for,
haskell	main, import, data, do, int, io, list, print, if, in, string, map, let, _, of	=, (,), ., „ [, main,], ->, $, \", ::, import, :, -,
icon	end, main, procedure, if, to, write, do, then, return, every, of, the, else, for, and	(,), end, „ main, :=, procedure, (), if, [, #, \", to, {, do,
j	if, end, do, _, nb, the, for, of, and, to, in, require, define, is, _1	.,), (, „ :, =:, -, +, =, *, \", ',], /, ;,

continua nella prossima pagina

Tabella 4.1 – continuo della pagina precedente

linguaggio	**tokens (no caratt. speciali)**	**tokens (con caratt. speciali)**
java	string, main, void, for, return, new, public, int, static, if, out, class, println, args, system	(, ., },), =, {, ;, „ string,);, main, for, return, void, int,
javascript	return, function, var, if, for, length, array, the, math, map, of, string, to, new, in	(, ., „ =,), {, ;, [, },);, return, function, :, +, -,
jq	def, if, as, end, then, else, the, length, and, of, range, in, is, to, reduce	(,), ., |, :, ;, „ -, $, [,], \", def, #, if,
julia	end, for, in, function, println, if, return, the, using, length, int, of, is, while, string	(,), =, „ end, :, ., [, for, \", -, in, function,], (\",
kotlin	string, main, array, println, fun, args, val, if, in, for, version, int, var, return, else	(, ., :,), {, }, =, <, string, main, „ array, println, args, fun,
lua	end, print, for, function, if, do, return, then, local, in, and, string, or, to, the	(,), =, „ ., end, print, for, function, if, \", +, do, return, [,
mathematica	if, print, range, length, for, module, true, table, in, the, and, while, is, do, of	„ [,], =, {, (, -, \", ., :=, ;,), +, \"„],„
nim	for, in, var, if, import, echo, proc, int, let, string, else, result, len, and, while	=, (, „ :,), ., \", for, [, var, in, import, if,], -,
ocaml	let, in, if, then, list, for, printf, else, to, fun, string, _, with, and, do	=, (, .,), let, \", in, ;, -, „ ->, if, :, (), string,
pari-gp	if, for, my, print, return, while, vector, the, str, sum, of, random, to, vec, in	(, „ =,),);, if, [, -, for, +, ;, my, ., print, (\",

continua nella prossima pagina

Tabella 4.1 – continuo della pagina precedente

linguaggio	**tokens (no caratt. speciali)**	**tokens (con caratt. speciali)**
perl	print, my, _, for, use, if, sub, return, the, while, and, to, is, map, shift	=, ;, (, „ $, {,), }, \", print,);, my, ., -, for,
perl-6	say, my, for, if, sub, _, is, to, map, return, the, new, int, and, of	., ;, =, {, „), }, (, $, say, -, :, for, my, \",
phix	end, if, for, do, to, integer, then, function, return, length, sequence, string, printf, and, constant	(, =,), „ end, [, if, to, for, do, integer, ., then,], -,
picolisp	for, let, if, de, and, nil, in, inc, do, println, setq, the, car, length, while	(,),)), \", ., for, let, ', "),))), :, #, de, -, ((,
powershell	if, function, _, foreach, in, get, for, object, string, else, the, return, and, eq, int	-,), $, =, }, ., {, „ (, [, \", if, |,], ($,
purebasic	if, to, for, procedure, print, str, the, input, endif, define, and, of, endprocedure, while, in	(, =,), „ ., if, +, (), to, \", ;, :, -, for, (\",
python	for, in, if, print, def, return, import, else, range, the, and, of, from, to, len	(, „), =, ., :, for, in, if, [, -, print,):, def,],
r	function, if, in, for, print, the, length, else, is, true, return, to, of, as, false	(,), „ ., =, {, }, <-, function, :, \", -,)), [, if,
rexx	in, the, if, to, do, for, end, say, of, then, it, return, we, all, program	=, (, ., „), :, to, in, the, if, \", do, /*, -, say,
racket	define, for, lang, in, racket, if, list, the, string, and, to, let, require, of, range	(,), -, #,)), \", define, /, for,))), lang, in, [, ., racket,
ring	for, to, if, see, nl, next, func, len, return, ok, string, and, while, list, project	=, (,), +, \", „ for, to, if, -, :, see, [,], .,

continua nella prossima pagina

Tabella 4.1 – continuo della pagina precedente

linguaggio	**tokens (no caratt. speciali)**	**tokens (con caratt. speciali)**
ruby	end, def, do, if, puts, new, the, each, class, in, for, and, of, is, to	., (,), =, „ end, \", [, |, :, #, }, def, -,],
rust	main, let, fn, println, for, in, if, use, std, new, the, _, mut, string, to	}, {, (, =, „ (), ., :,), main, ;, fn, let,);, println,
scala	def, val, if, println, int, _, string, to, for, object, import, else, new, case, map	(,), =, ., :, „ {, }, def, \", [, println, val, +, if,
sidef	var, say, func, for, if, in, return, each, len, of, map, the, join, print, to	(,), ., =, {, }, „ var, say, [, -, ;, \", :, func,
swift	let, in, for, return, var, func, int, print, if, println, string, import, count, else, _	(,), =, {, }, :, „ ., let, in, for, var, [, \", return,
tcl	set, if, puts, for, return, proc, the, list, to, foreach, expr, of, package, string, require	[, {, }, $,], \", set, -, ., if, puts, return, #, for, (,
zkl	println, if, in, fcn, list, return, var, foreach, and, of, len, fmt, the, to, while	(, ., „), },);, =, :=, (\", ;, [, println,){, if, ();,

Tabella 4.1: Lista dei tokens più "rilevanti" ogni linguaggio.

Presa questa lista, concateniamo tutti i tokens $T = (t1, \ldots, t_m)$ di tutti i linguaggi $C = (c_1, \ldots, c_n)$ ottenendo un lista L come segue:

$$L = \{l_{i,j} \mid i \in [1, m] \wedge j \in [1, n]\}$$

Una volta ottenuta questo, per ogni file sorgente formato da m tokens $T = (t1, \ldots, t_m)$ e appartenente ad una classe $c_x \in C$, le ***features*** sono formate da una lista F di valori costruita come segue:

$$F = \{f_{1,1}, ..., f_{1,j}, ..., fi, 1, ..., f_{i,j}\} \quad \text{dove} \quad f_{i,j} \in \{0, 1\}$$

dove ciascun elemento è definito come:

$$f_{i,j} = \begin{cases} 1 & \text{se } t_i = L_{i,j} \\ 0 & \text{se } altrimenti \end{cases} \qquad \forall i \in [1, m], j \in [1, n]$$

Essendo il nostro dataset (*training-set*) formato da 48 linguaggi, il numero di ***features*** per ogni file di sorgente è pari a 720 (15 tokens per ogni linguaggio).

Capitolo 5

Rete Neurale

In questo capitolo descriviamo l'applicazione di un classificatore e di una parte della procedura di estrazione delle features già utilizzati in una ricerca precedente [4] basata sul riconoscimento dell'estensione di un file di codice sorgente attraverso l'uso di un classificatore implementato attraverso una rete neurale.

Questo problema rappresenta un punto di partenza rispetto a quello trattato in questo elaborato, in quanto, tranne che per alcune eccezioni, esiste una corrispondenza 1 a 1 tra estensione e linguaggio di programmazione.

5.1 Background

Le reti neurali sono reti formate da nodi o unità (vedi Figura 5.1) connesse attraverso archi chiamati 'collegamenti'. Un collegamento dall'unità i all'unità j è costruito con lo scopo di propagare l'attivazione a_i da i a j. Ogni collegamento ha anche un peso $w_{i,j}$ associato ad esso, il quale rappresenta il 'valore' e il 'segno' della connessione. Proprio come nei modelli di regressione lineare, ogni unità ha un input fittizio $a_0 = 1$ con un peso associato $w_{0,j}$ [15].

Ogni unità j prima calcola una somma ponderata dei suoi input (i pesi sono i parametri appresi che rappresentano le connessioni):

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i$$

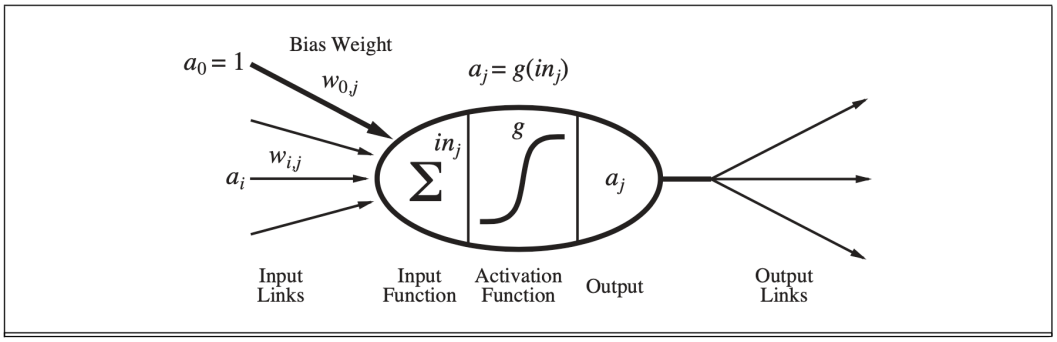

Figura 5.1: Modello matematico di un 'neurone' ideato da McCulloch e Pitts (1943).

e, successivamente, applica una funzione di attivazione g a questa somma ponderata ottenendo in output:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i)$$

Queste 'unità' logiche alla base della rete prendono il nome di **percettori**.

Una volta deciso il modello matematico per i singoli 'neuroni', il prossimo passo è capire come collegarli insieme al fine di formare una rete. In base a come viene eseguito questo processo di collegamento, otterremo una rete appartenente a una di queste due macro-categorie:

- Le reti *feed-forward* hanno connessioni verso una sola direzione, formando così un grafico aciclico diretto. In sostanza, ogni nodo riceve informazioni di input da nodi del "livello superiore" e invia l'output generato ai nodi del "livello inferiore" (in questo modo non si formano cicli/anelli). In altre parole, questa tipologia di rete rappresenta una funzione del suo input corrente e, di conseguenza, non possiede uno stato interno al di fuori dei suoi stessi pesi [15].

- Una rete *recurrent* (ricorrente), invece, re-immette l'output dei propri nodi come input degli stessi. Questo significa che i livelli di attivazione della rete formano un sistema dinamico che può raggiungere uno proprio 'stato', il quale può essere più o meno 'stabile'. Inoltre, la risposta di questo tipo di rete a un determinato input dipende, quindi, dal suo stato iniziale, il quale, a sua volta, può dipendere da input precedenti [15].

5.2 Struttura

Il modello selezionato è una rete neurale feed-forward (Sezione 5.1) che dispone di 48 'unità' di output, le quali corrispondono al numero di possibili linguaggi che consideriamo. La struttura complessiva è simile a un codificatore, in quanto la 'dimensionalità' iniziale dell'input viene ridotta durante il passaggio tra i vari livelli della rete [4].

Oltre ai livelli di input e output, la rete è composta da 3 livelli 'Dense' (nascosti) costruiti rispettivamente con 1000, 800 e 700 unità. Ad ogni livello di questo tipo viene aggiunto, immediatamente dopo, un layer di 'Dropout' (con un tasso di riduzione pari a 0.5).

Dense Layer. Questa tipologia di layer implementa l'operazione: $out = act(dot(in, kernel) + bias)$ dove in è l'input, act è la funzione di attivazione passata come argomento, dot rappresenta l'operazione 'prodotto scalare', $kernel$ è una matrice di pesi (weights) creata da questo layer [17]. Infine, $bias$ è un vettore contenente un ulteriore set di pesi (weights), i quali corrispondono all'output di una rete neurale quando non ha input. In sostanza, questo rappresenta un neurone extra incluso in ogni strato di pre-output, il quale non è collegato a nessuno dei livelli precedenti della rete [18].

Dropout Layer. Il livello Dropout imposta casualmente le unità di input a 0 con una frequenza pari al ritmo ($rate$) di ogni 'step' durante la fase di 'training', in modo da prevenire l''overfitting'. Gli input non settati a 0 vengono scalati di un fattore pari a $1/(1-rate)$ in modo tale che la somma di tutti gli input rimanga invariata [19].

Poiché il processo di classificazione si basa su più classi, utilizziamo la funzione 'categorical cross entropy loss' [20] che viene spesso sfruttata in problemi con più classi di riconoscimento [4]. Nella fase di 'training', invece, abbiamo utilizzato l'ottimizzatore Adam [21] con un tasso di apprendimento pari a 0.0001, basandoci sui buoni risultati ottenuti nella ricerca prima citata [4]. Oltre ad aver mantenuto la stessa struttura (della rete) e le stesse funzioni di classificazione e di ottimizzazione di questa ricerca, abbiamo usato lo stesso numero di epoche di 'training' (8) in modo tale da poter paragonare in modo significativo i risultati ottenuti.

nome	**tipologia**	**output**
dense_1	Dense	(None, 1000)
dropout_1	Dropout	(None, 1000)
dense_2	Dense	(None, 800)
dropout_2	Dropout	(None, 800)
dense_3	Dense	(None, 700)
dropout_3	Dropout	(None, 700)
dense_4	Dense	(None, 48)

Tabella 5.1: Struttura della Rete Neurale.

5.3 Estrazione delle Features

Prima di procedere all'estrazione delle features, ricorsivamente, per ogni linguaggio, accediamo al contenuto di ogni file sorgente presente nel training-set, rimuovendo tutti i tokens che non compaiono all'interno di almeno 1 esempio (su 400) della classe a cui appartiene.

A questo punto possiamo procedere creando, per ogni esempio, la lista delle **features**: preso in input un file, sostituiamo ogni token del nostro vocabolario (generato nella sezione 2.3) con un valore corrispondente alla frequenza con cui esso cui compare all'interno di questo specifico snippet. In questo modo ogni sorgente del dataset conterrà lo stesso numero di features, il quale corrisponderà al numero totale di tokens all'interno del nostro vocabolario.

5.4 Esempio

Consideriamo un file di codice sorgente scritto nel linguaggio "Python":

```
def partitions __a__
    partitions __a__ append __n__
    for __a__ in xrange __n__ __a__ __n__
        __a__ __a__ __a__ __n__ __a__ __n__ __n__
        if __a__ __n__
        if __a__ __n__
```

```
            partitions __a__ __a__ partitions __a__ __a__
        else
            partitions __a__ __a__ partitions __a__ __a__
        __a__ __a__
        if __a__ __n__

            if __a__ __n__
                partitions __a__ __a__ partitions __a__ __a__
            else
                partitions __a__ __a__ partitions __a__ __a__
    return partitions __a__ __n__

partitions __a__ __n__

def main
    ns set __n__ __n__ __n__ __n__
    max ns max ns
    for __a__ in xrange __n__ max ns __n__
        if __a__ max ns
            __a__ partitions __a__
        if __a__ in ns
            print 6d __a__ __a__ __a__

main
```

Snippet 5.1: Rete Neurale: esempio di un file sorgente.

e consideriamo un **vocabolario** ipotetico:

```
{
    'def': 1,           'return': 2,        'partitions': 3,
    'str': 4,           'append': 5,        'raise': 6,
    'for': 7,           'of': 8,            'in': 9,
    'xrange': 10,       'main': 11,         'include': 12,
    'ns': 13,           'max': 14,          'from': 15,
    'print': 16,        'if': 17,           'dict': 18,
    'normalize': 19,    'global': 20
}
```

Snippet 5.2: Rete Neurale: esempio di vocabolario.

e la lista di **features** generate sarà:

```
[ 2/47, 1/47, 13/47, 0, 1/47, 0, 2/47, 0, 3/47, 2/47, 2/47, 0, 6/47, 4/47, 0, 1/47,
    6/47, 0, 0, 0 ]
```

Snippet 5.3: Rete Neurale: esempio di features.

Capitolo 6

Risultati Sperimentali

Introduciamo le definizioni delle metriche usate per la valutazione dei risultati ottenuti.

6.1 Metriche

Definizione (Precisione)**.** *La 'Precisione' (Precision), per una classe, è definita come il numero di veri positivi (ovvero il numero di oggetti etichettati correttamente come appartenenti alla classe) diviso il numero totale di elementi etichettati come appartenenti alla classe (la somma di veri positivi e falsi positivi, che sono oggetti etichettati correttamente/erroneamente come appartenenti alla classe) [22].*

$$Precision = \frac{Veri\_Positivi}{Veri\_Positivi + Falsi\_Positivi}$$

Definizione (Recupero)**.** *Il 'Recupero' (Recall) è definito come il numero di veri positivi diviso il numero totale di elementi che effettivamente appartengono alla classe [22].*

$$Recall = \frac{Veri\_Positivi}{Veri\_Positivi + Falsi\_Negativi}$$

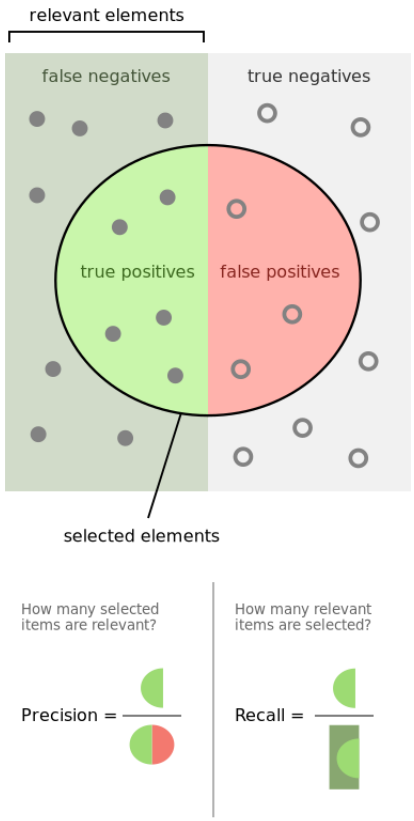

Figura 6.1: Differenza tra la metriche precision and recall [2].

Definizione (F1-Score)**.** *L' "F1-Score" (noto anche come "F-Score" o "F-measure") è una misura dell'accuratezza di un test. Questa misura è calcolata tramite la media armonica di Precisione e Recupero [2]:*

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} = 2 * \frac{precision * recall}{precision + recall}$$

La metrica "F1-Score" viene anche utilizzata per valutare i problemi di classificazione con più di due classi (come nel nostro caso). In questo caso, il valore finale è ottenuto mediante l'operazione di "weighted-averaging" o "macro-averaging" (per maggiori dettagli [2]).

6.2 Parametri di Confronto

Prima di procedere al confronto con i risultati ottenuti nelle altre ricerche, analizziamo come la presenza o meno dei caratteri speciali influisca sulla percentuale finale di riconoscimento.

Caratteri Speciali Confrontando i valori della metrica f1-score (che indica, in generale, l'accuratezza della classificazione) notiamo come, per tutti i modelli usati, la percentuale di classi identificate correttamente sia maggiore mantenendo questi caratteri, percentuale che appunto tende a diminuire in modo significativo (fino al 10%) eliminandoli.

modello	con speciali	senza speciali	Δ f1-score
Naive Bayes	0.88	0.78	0.10
SVM	0.80	0.74	0.06
Rete Neurale	0.83	0.73	0.10

Tabella 6.1: Confronto risultati con e senza caratteri speciali.

Questo risultato dimostra, a primo impatto, un aspetto piuttosto scontato: per risolvere in modo concreto il problema che stiamo affrontando, non possiamo prescindere dai caratteri speciali all'interno dei file sorgenti, i quali contengono, in modo intrinseco, una fortissima componente identificativa dello stesso linguaggio.

Una volta chiarito questo aspetto, procediamo confrontando (con le ricerche precedenti), i risultati ottenuti considerando solo i test effettuati (all'interno di questo progetto) mantenendo i caratteri speciali, in quanto, come abbiamo spiegato nel paragrafo precedente, producono un'accuratezza migliore nella classificazione.

Ricerche Precedenti Confrontiamo i risultati della metrica f1-score tenendo presente la dimensione del dataset e il numero di linguaggi considerati in ogni ricerca.

	ricerca precedente			questo progetto		
modello	classi	esempi	f1-score	classi	esempi	f1-score
Naive Bayes [9]	10	$\approx$ 2K	0.93	48	$\approx$ 40K	0.88
SVM [5]	10	$\approx$ 300	0.89	48	$\approx$ 40K	0.80
Rete Neurale [4]	121	$\approx$ 2M	0.85	48	$\approx$ 40K	0.83

Tabella 6.2: Confronto risultati con ricerche precedenti.

I risultati ottenuti vanno contestualizzati in base al modello considerato:

- **Naive Bayes**: questo classificatore, grazie alla sua 'propensione' per i problemi relativi al riconoscimento del testo e alle sperimentazione di un 'nuovo' (non presente in altre ricerche) processo di selezione delle features, ha generato i risultati migliori ($\approx 88\%$) all'interno di questo progetto. La ricerca [9] ha ottenuto una percentuale di riconoscimento maggiore ($\approx 93\%$) ma lavorando su un $1/4$ delle classi e su un dataset 20 volte più piccolo.

- **Support Vector Machine**: usando questo classificatore, abbiamo ereditato da una ricerca anche il processo di selezione delle features, modellando alcuni parametri rispetto al nostro dataset. Nonostante una fase di pre-processamento differente, abbiamo ottenuto una considerevole ($\approx 10\%$) differenza nell'accuratezza finale, a dimostrazione del fatto di come questa rappresenti una 'soglia limite' che, per il momento, non si riesca ad oltrepassare. La ricerca [5] ha ottenuto una percentuale di riconoscimento pari a ($\approx 89\%$) ma lavorando su un $1/4$ delle classi e su un dataset ≈ 100 volte più piccolo.

- **Rete Neurale**: questo classificatore, nonostate sia stato ereditato da una ricerca su un problema simile (ma non uguale) al nostro e sia stato costruito basandosi su un processo di selezione delle features profondamente diverso, ha ottenuto percentuali di classificazione in linea con quelle originali. Infatti, nella ricerca in questione [4] ha generato una percentuale di classificazione leggermente maggiore ($\approx$85%) lavorando, però, su un numero di classi pari a ≈ 3 volte quelle considerate da noi e su un dataset 50 volte più grande del nostro.

- **Altre**: nella Tabella 6.2 non abbiamo inserito alcune delle ricerche citate nella Tabella 1.2 considerando le notevoli differenze presenti sia nella scelta del classificatore, sia nel processo di selezione delle features. Nonostante questo, è possibile confrontare in modo superficiale alcuni risultati con i nostri, evidenziando come alcune tra queste, in particolare [10], [11] e [8], abbiano ottenuto percentuali di riconoscimento rispettivamente pari a $\approx$ 96%, $\approx 99\%$ e $\approx 97\%$, le quali sono significativamente migliori rispetto alle nostre. Chiaramente quest'ultimo aspetto deriva dall'applicazione di una fase di pre-processamento selezione delle features profondamente diversa dalla nostra.

Questi risultati, oltre che ad indirizzarci verso classificatori che, a primo impatto, rispetto ai nostri, sembrano ottenere 'performance' di classificazione migliori, indicano come non sia conveniente considerare *tokens* esclusivamente 1-gram, come nel nostro caso. Quest'ultimo aspetto è giustificato anche dalle proprietà derivate dalla sintassi dei linguaggi di programmazione: la

maggior parte dei costrutti sintattici vengono identificati attraverso una sequenza di 'parole chiave' all'interno di ciascun linguaggio. Considerando solo *tokens* 1-gram rinunciamo a 'catturare' le proprietà identificative di queste sequenze di keyword, le quali sono alla base della costruzione dello stesso linguaggio.

6.3 Reports di Classificazione

Di seguito riportiamo tutti i reports di classificazione dettagliati con indicate le tre metriche *Precision* (P), *Recall* (R) e *F1-score* (F1) per ciascuna classe. Inoltre, calcoliamo per ogni *Metrica* (tra quelle appena elencate) anche un valore Δ definito come:

$$\Delta = Metrica_{(con\_speciali)} - Metrica_{(senza\_speciali)}$$

Questo valore ci aiuterà a stimare il miglioramento/peggioramento di una determinata metrica all'interno dei due test da effettuati in questo elaborato.

Multinomial Naive Bayes

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
ada	0.79	0.98	0.88	0.42	0.94	0.58	0.37	0.04	0.30
algol-68	0.79	0.96	0.87	0.53	0.96	0.68	0.26	0.00	0.19
autohotkey	0.87	0.92	0.90	0.80	0.86	0.83	0.07	0.06	0.07
awk	0.43	0.92	0.59	0.36	0.85	0.50	0.07	0.07	0.09
bbc-basic	0.83	0.96	0.89	0.51	0.86	0.64	0.32	0.10	0.25
c	0.87	0.90	0.88	0.83	0.86	0.85	0.04	0.04	0.03
c++	0.90	0.92	0.91	0.93	0.88	0.90	-0.03	0.04	0.01
clojure	0.91	0.86	0.89	0.79	0.78	0.78	0.12	0.08	0.11
common-lisp	0.94	0.90	0.92	0.90	0.81	0.85	0.04	0.09	0.07
d	0.95	0.97	0.96	0.94	0.96	0.95	0.01	0.01	0.01
elixir	0.71	0.91	0.80	0.52	0.84	0.64	0.19	0.07	0.16
erlang	0.73	0.91	0.81	0.65	0.86	0.74	0.08	0.05	0.07
factor	0.86	0.86	0.86	0.76	0.76	0.76	0.10	0.10	0.10
forth	0.75	0.91	0.82	0.71	0.85	0.77	0.04	0.06	0.05

continua nella prossima pagina

Tabella 6.3 – continuo della pagina precedente

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
fortran	0.75	0.90	0.82	0.60	0.85	0.70	0.15	0.05	0.12
freebasic	0.52	1.00	0.68	0.26	0.94	0.41	0.26	0.06	0.27
go	0.88	0.93	0.91	0.81	0.93	0.86	0.07	0.00	0.05
haskell	0.95	0.84	0.89	0.93	0.78	0.85	0.02	0.06	0.04
icon	0.72	0.95	0.82	0.58	0.92	0.71	0.14	0.03	0.11
j	0.98	0.70	0.82	0.98	0.33	0.49	0.00	0.37	0.33
java	0.91	0.90	0.91	0.84	0.90	0.87	0.07	0.00	0.04
javascript	0.91	0.80	0.85	0.85	0.77	0.81	0.06	0.03	0.04
jq	0.89	0.81	0.85	0.85	0.70	0.77	0.04	0.11	0.08
julia	0.91	0.84	0.87	0.86	0.71	0.78	0.05	0.13	0.09
kotlin	0.91	0.99	0.95	0.78	0.98	0.87	0.13	0.01	0.08
lua	0.65	0.88	0.75	0.58	0.84	0.69	0.07	0.04	0.06
mathematica	0.83	0.83	0.83	0.86	0.40	0.55	-0.03	0.43	0.28
nim	0.84	0.93	0.89	0.75	0.91	0.82	0.09	0.02	0.07
ocaml	0.90	0.88	0.89	0.91	0.81	0.86	-0.01	0.07	0.03
pari-gp	0.77	0.73	0.75	0.63	0.52	0.57	0.14	0.21	0.18
perl	0.88	0.88	0.88	0.80	0.81	0.81	0.08	0.07	0.07
perl-6	0.90	0.87	0.88	0.84	0.78	0.81	0.06	0.09	0.07
phix	0.96	0.93	0.94	0.87	0.86	0.86	0.09	0.07	0.08
picolisp	0.89	0.88	0.88	0.92	0.76	0.83	-0.03	0.12	0.05
powershell	0.85	0.83	0.84	0.77	0.66	0.71	0.08	0.17	0.13
purebasic	0.86	0.93	0.89	0.75	0.85	0.80	0.11	0.08	0.09
python	0.96	0.86	0.91	0.89	0.76	0.82	0.07	0.10	0.09
r	0.80	0.80	0.80	0.61	0.52	0.56	0.19	0.28	0.24
racket	0.90	0.91	0.90	0.83	0.91	0.87	0.07	0.00	0.03
rexx	0.90	0.95	0.93	0.80	0.94	0.86	0.10	0.01	0.07
ring	0.63	0.96	0.76	0.56	0.91	0.69	0.07	0.05	0.07
ruby	0.87	0.87	0.87	0.86	0.74	0.79	0.01	0.13	0.08
rust	0.89	0.94	0.91	0.75	0.94	0.83	0.14	0.00	0.08
scala	0.89	0.92	0.90	0.88	0.88	0.88	0.01	0.04	0.02
sidef	0.81	0.89	0.85	0.76	0.82	0.79	0.05	0.07	0.06
swift	0.66	0.92	0.77	0.48	0.84	0.61	0.18	0.08	0.16
tcl	0.97	0.92	0.94	0.88	0.87	0.87	0.09	0.05	0.07

continua nella prossima pagina

Tabella 6.3 – continuo della pagina precedente

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
zkl	0.93	0.93	0.93	0.92	0.81	0.86	0.01	0.12	0.07
accuracy			**0.88**			**0.78**			**0.10**
macro avg	0.84	0.90	0.86	0.75	0.81	0.76	0.09	0.09	0.10
weighted avg	0.89	0.88	0.88	0.82	0.78	0.78	0.07	0.10	0.10

Tabella 6.3: Risultati ottenuti con il classificatore Naive-Bayes.

SVM

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
ada	0.95	0.95	0.95	0.92	0.86	0.89	0.03	0.09	0.06
algol-68	0.70	0.79	0.74	0.55	0.85	0.66	0.15	-0.06	0.08
autohotkey	0.61	0.80	0.69	0.70	0.78	0.74	-0.09	0.02	-0.05
awk	0.42	0.79	0.55	0.57	0.85	0.69	-0.15	-0.06	-0.14
bbc-basic	0.45	0.84	0.58	0.35	0.76	0.48	0.10	0.08	0.10
c	0.82	0.82	0.82	0.81	0.76	0.78	0.01	0.06	0.04
c++	0.81	0.86	0.83	0.79	0.85	0.82	0.02	0.01	0.01
clojure	0.66	0.83	0.73	0.74	0.66	0.70	-0.08	0.17	0.03
common-lisp	0.80	0.79	0.79	0.86	0.67	0.75	-0.06	0.12	0.04
d	0.91	0.90	0.90	0.97	0.93	0.95	-0.06	-0.03	-0.05
elixir	0.50	0.82	0.62	0.60	0.74	0.66	-0.10	0.08	-0.04
erlang	0.47	0.87	0.61	0.73	0.83	0.77	-0.26	0.04	-0.16
factor	0.55	0.78	0.64	0.60	0.62	0.61	-0.05	0.16	0.03
forth	0.57	0.78	0.66	0.61	0.67	0.63	-0.04	0.11	0.03
fortran	0.71	0.82	0.76	0.70	0.73	0.72	0.01	0.09	0.04
freebasic	0.41	1.00	0.58	0.35	0.94	0.51	0.06	0.06	0.07
go	0.95	0.88	0.91	0.91	0.88	0.90	0.04	0.00	0.01
haskell	0.86	0.74	0.80	0.80	0.70	0.75	0.06	0.04	0.05
icon	0.71	0.84	0.77	0.71	0.84	0.77	0.00	0.00	0.00

continua nella prossima pagina

Tabella 6.4 – continuo della pagina precedente

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
j	0.78	0.78	0.78	0.46	0.80	0.58	0.32	-0.02	0.20
java	0.90	0.81	0.86	0.86	0.86	0.86	0.04	-0.05	0.00
javascript	0.82	0.70	0.75	0.85	0.68	0.76	-0.03	0.02	-0.01
jq	0.78	0.76	0.77	0.69	0.58	0.63	0.09	0.18	0.14
julia	0.80	0.73	0.76	0.79	0.62	0.69	0.01	0.11	0.07
kotlin	0.93	0.95	0.94	0.93	0.97	0.95	0.00	-0.02	-0.01
lua	0.71	0.81	0.76	0.68	0.81	0.74	0.03	0.00	0.02
mathematica	0.67	0.78	0.72	0.34	0.31	0.33	0.33	0.47	0.39
nim	0.70	0.80	0.75	0.71	0.90	0.80	-0.01	-0.10	-0.05
ocaml	0.83	0.79	0.81	0.72	0.77	0.75	0.11	0.02	0.06
pari-gp	0.34	0.69	0.46	0.37	0.50	0.43	-0.03	0.19	0.03
perl	0.82	0.74	0.78	0.75	0.70	0.72	0.07	0.04	0.06
perl-6	0.83	0.73	0.78	0.78	0.69	0.73	0.05	0.04	0.05
phix	0.92	0.83	0.87	0.85	0.74	0.79	0.07	0.09	0.08
picolisp	0.85	0.75	0.80	0.70	0.61	0.65	0.15	0.14	0.15
powershell	0.66	0.75	0.70	0.59	0.52	0.55	0.07	0.23	0.15
purebasic	0.56	0.76	0.65	0.75	0.65	0.70	-0.19	0.11	-0.05
python	0.93	0.74	0.82	0.93	0.69	0.79	0.00	0.05	0.03
r	0.70	0.73	0.71	0.37	0.47	0.41	0.33	0.26	0.30
racket	0.95	0.83	0.88	0.89	0.81	0.85	0.06	0.02	0.03
rexx	0.96	0.85	0.90	0.91	0.83	0.87	0.05	0.02	0.03
ring	0.75	0.89	0.82	0.78	0.87	0.82	-0.03	0.02	0.00
ruby	0.86	0.77	0.81	0.88	0.67	0.76	-0.02	0.10	0.05
rust	0.83	0.88	0.85	0.85	0.85	0.85	-0.02	0.03	0.00
scala	0.93	0.81	0.87	0.88	0.82	0.85	0.05	-0.01	0.02
sidef	0.84	0.78	0.81	0.77	0.73	0.75	0.07	0.05	0.06
swift	0.64	0.83	0.73	0.51	0.81	0.62	0.13	0.02	0.11
tcl	0.94	0.83	0.88	0.91	0.85	0.88	0.03	-0.02	0.00
zkl	0.86	0.75	0.80	0.85	0.72	0.78	0.01	0.03	0.02
accuracy			**0.80**			**0.74**			**0.06**
macro avg	0.75	0.81	0.77	0.72	0.74	0.72	0.03	0.07	0.05

continua nella prossima pagina

Tabella 6.4 – continuo della pagina precedente

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
weighted avg	0.82	0.80	0.80	0.77	0.74	0.74	0.05	0.06	0.06

Tabella 6.4: Risultati ottenuti con il classificatore SVM.

Rete Neurale

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
ada	0.94	0.94	0.94	0.87	0.90	0.89	0.07	0.04	0.05
algol-68	0.75	0.88	0.81	0.48	0.89	0.62	0.27	-0.01	0.19
autohotkey	0.83	0.91	0.87	0.73	0.84	0.78	0.10	0.07	0.09
awk	0.38	0.81	0.52	0.62	0.81	0.70	-0.24	0.00	-0.18
bbc-basic	0.73	0.92	0.81	0.45	0.72	0.55	0.28	0.20	0.26
c	0.86	0.83	0.84	0.83	0.72	0.77	0.03	0.11	0.07
c++	0.86	0.87	0.86	0.84	0.85	0.84	0.02	0.02	0.02
clojure	0.80	0.80	0.80	0.61	0.76	0.67	0.19	0.04	0.13
common-lisp	0.89	0.82	0.86	0.73	0.79	0.76	0.16	0.03	0.10
d	0.95	0.94	0.94	0.94	0.91	0.92	0.01	0.03	0.02
elixir	0.56	0.85	0.67	0.51	0.72	0.60	0.05	0.13	0.07
erlang	0.61	0.85	0.71	0.44	0.77	0.56	0.17	0.08	0.15
factor	0.68	0.85	0.75	0.58	0.76	0.66	0.10	0.09	0.09
forth	0.65	0.84	0.73	0.56	0.79	0.65	0.09	0.05	0.08
fortran	0.78	0.86	0.82	0.70	0.77	0.73	0.08	0.09	0.09
freebasic	0.40	1.00	0.57	0.50	0.90	0.64	-0.10	0.10	-0.07
go	0.93	0.90	0.92	0.86	0.89	0.88	0.07	0.01	0.04
haskell	0.86	0.74	0.79	0.83	0.71	0.77	0.03	0.03	0.02
icon	0.79	0.89	0.83	0.73	0.83	0.78	0.06	0.06	0.05
j	0.87	0.73	0.79	0.63	0.50	0.56	0.24	0.23	0.23
java	0.89	0.88	0.89	0.84	0.85	0.85	0.05	0.03	0.04
javascript	0.84	0.78	0.81	0.79	0.67	0.73	0.05	0.11	0.08
jq	0.86	0.76	0.80	0.78	0.66	0.71	0.08	0.10	0.09

continua nella prossima pagina

Tabella 6.5 – continuo della pagina precedente

	con speciali			senza speciali			Δ		
linguaggio	P	R	F1	P	R	F1	P	R	F1
julia	0.73	0.73	0.73	0.73	0.61	0.67	0.00	0.12	0.06
kotlin	0.89	0.91	0.90	0.83	0.87	0.85	0.06	0.04	0.05
lua	0.63	0.81	0.71	0.65	0.69	0.67	-0.02	0.12	0.04
mathematica	0.75	0.83	0.79	0.50	0.41	0.45	0.25	0.42	0.34
nim	0.72	0.88	0.79	0.69	0.81	0.74	0.03	0.07	0.05
ocaml	0.86	0.83	0.85	0.86	0.76	0.81	0.00	0.07	0.04
pari-gp	0.66	0.78	0.72	0.34	0.64	0.44	0.32	0.14	0.28
perl	0.87	0.74	0.80	0.77	0.65	0.70	0.10	0.09	0.10
perl-6	0.75	0.78	0.77	0.67	0.73	0.70	0.08	0.05	0.07
phix	0.85	0.92	0.88	0.82	0.71	0.76	0.03	0.21	0.12
picolisp	0.85	0.82	0.83	0.67	0.73	0.70	0.18	0.09	0.13
powershell	0.71	0.81	0.76	0.51	0.58	0.54	0.20	0.23	0.22
purebasic	0.81	0.87	0.84	0.74	0.84	0.79	0.07	0.03	0.05
python	0.91	0.75	0.82	0.78	0.65	0.71	0.13	0.10	0.11
r	0.61	0.76	0.67	0.38	0.48	0.42	0.23	0.28	0.25
racket	0.91	0.87	0.89	0.83	0.86	0.85	0.08	0.01	0.04
rexx	0.94	0.93	0.93	0.89	0.90	0.89	0.05	0.03	0.04
ring	0.62	0.93	0.74	0.56	0.84	0.67	0.06	0.09	0.07
ruby	0.79	0.77	0.78	0.68	0.62	0.65	0.11	0.15	0.13
rust	0.83	0.89	0.86	0.67	0.88	0.76	0.16	0.01	0.10
scala	0.82	0.84	0.83	0.76	0.77	0.76	0.06	0.07	0.07
sidef	0.74	0.78	0.76	0.66	0.73	0.69	0.08	0.05	0.07
swift	0.49	0.85	0.62	0.43	0.74	0.55	0.06	0.11	0.07
tcl	0.91	0.91	0.91	0.90	0.78	0.84	0.01	0.13	0.07
zkl	0.94	0.87	0.91	0.83	0.75	0.79	0.11	0.12	0.12
accuracy			**0.83**			**0.73**			**0.10**
macro avg	0.78	0.84	0.80	0.69	0.75	0.71	0.09	0.09	0.09
weighted avg	0.84	0.83	0.83	0.74	0.73	0.73	0.10	0.10	0.10

Tabella 6.5: Risultati ottenuti con un classificatore basato su Rete Neurale.

Conclusioni

Questo Elaborato. In questo progetto abbiamo affrontato un problema riguardante la costruzione di un classificatore in grado di identificare, in modo automatico, il linguaggio di programmazione di un file generico di codice sorgente, senza usare euristiche o informazioni a priori (es: l'estensione del file). Questo progetto segue la linea dettata da altre ricerche (sullo stesso problema), partendo dai risultati ottenuti in queste.

Inoltre, sottolineiamo come i risultati ottenuti potrebbero essere usati sia per classificare ogni frammento di codice non classificato sul Web (piattaforme cloud di version-control, archivi legacy, forum ...), sia come strumento di supporto per gli editor testuali (correzioni, evidenziazione della sintassi ...).

Per risolvere questo problema abbiamo, prima di tutto, costruito una 'nuova' (rispetto alle ricerche menzionante) fase di pre-processamento e l'abbiamo applicata a 3 classificatori diversi seguendo diverse logiche:

- nel primo caso, abbiamo usato un classificatore Support Vector Machine, usando un fase di selezione delle features descritta nella ricerca [5].
- nel secondo caso, abbiamo usato un classificatore basato su una Rete Neurale usato nella ricerca [4], a cui abbiamo applicato una 'diversa' (rispetto all'originale) fase di selezione delle features attraverso l'uso del vocabolario generato dalla nostra fase di pre-processamento.
- nel terzo caso, abbiamo utilizzato un classificatore molto comune nei problemi di riconoscimento del testo (presente nella ricerca [9]) applicando una 'nuova' fase di selezione delle features (non presente in nessuna delle ricerche citate).

Come supporto a ciascuna delle fasi di selezione delle features appena menzionate, abbiamo usato il vocabolario generato da una 'nuova' fase di pre-processamento. Questa fase è stata

costruita con l'obiettivo di distaccarsi dall'approccio usato negli elaborati originali, selezionando esclusivamente *tokens* 1-gram e rimuovendo commenti, numeri e caratteri alfanumerici unari. Tutte queste fasi, compresi i classificatori, sono state applicate in due esperimenti: il primo mantenendo i caratteri speciali, il secondo eliminandoli.

Abbiamo applicato il processo sopra descritto a un dataset [6] non molto grande ($\approx$ 40K files), lavorando sul riconoscimento di un totale di 48 linguaggi. Rispetto alle ricerche originali non abbiamo ottenuto risultati di classificazioni 'rilevanti', anche se dobbiamo considerare le semplificazioni significative adottate nella fase di pre-processamento (es: selezione di tokens unicamente 1-gram), le quali ipotizziamo abbiamo inciso in modo significativo sulla generazione del vocabolario, e, di conseguenza, sulle 'performance' di classificazione.

Sviluppi Futuri. A seguito dei risultati in questo elaborato, e da quelli ottenuti nelle precedenti ricerche, ipotizziamo che la generazione di un vocabolario basato su tokens 2-gram possa costituire un punto chiave per il miglioramento dei risultati di classificazione. Chiaramente questo step aggiuntivo richiederebbe alcune modifiche alla fase di pre-processamento, all'interno della quale, esclusivamente per i tokens 2-gram, dovremmo evitare di eliminare i caratteri 'generici' "`__a__`" e "`__n__`" che abbiamo inserito. In questo modo, potremmo catturare in modo più preciso le proprietà sintattiche e grammaticali delle 'parole chiave' di ogni singolo linguaggio.

Un altro miglioramento potrebbe essere ottenuto attribuendo un'importanza maggiore ad alcuni tokens durante la fase di costruzione delle features. Infatti, basti pensare a come la presenza o la posizione di una determinata keyword (es: parentesi) rispecchi significativamente le proprietà di uno specifico linguaggio. Allo stesso modo però, potremmo diminuire 'l'importanza' di alcuni caratteri molto più generici (es: nomi delle variabili, operatori, ...).

Bibliografia

[1] Support vector machines - mathematical formulation. [Online]. Available: https://scikit-learn.org/stable/modules/svm.html#mathematical-formulation

[2] Wikipedia - f1 score. [Online]. Available: https://en.wikipedia.org/wiki/F1_score

[3] Wikipedia - github. [Online]. Available: https://en.wikipedia.org/wiki/GitHub

[4] M. G. e. S. Z. Francesca Del Bonifro, "Content-based file type identification for source code," Tech. Rep., 2020.

[5] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: Automatic classification of source code archives," *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, p. 632 – 638, 2002. [Online]. Available: http://doi.acm.org/10.1145/775047.775141

[6] Rosetta code. [Online]. Available: http://www.rosettacode.org/wiki/Rosetta_Code

[7] Rosetta code (github) repository. [Online]. Available: https://github.com/acmeism/RosettaCodeData

[8] S. Gilda, "Source code classification using neural networks," 2017.

[9] J. N. Khasnabish, M. Sodhi, J. Deshmukh, and G. Srini-vasaraghavan, "Detecting programming language from source code using bayesian learning techniques," *Machine Learning and Data Mining in Pattern Recognition*, p. 513–522, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08979-9_39

[10] J. K. v. Dam and V. Zaytsev, "Software language identification with natural language classifiers," *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, p. 624–628, 2016.

[11] S. Zevin and C. Holzem, "Machine learning based source code classification using syntax oriented features," 2017.

[12] Naive bayes. [Online]. Available: https://scikit-learn.org/stable/modules/naive_bayes.html

[13] Joachims, "T. text categorization with support vector machines," *In Proceedings of the Tenth European Conference on Machine Learnin*, pp. 137–142, 1999.

[14] K. J. T., "Automated text categorization using support vector machines," *Proceedings of the International Conference on Neural Information Processing*, pp. 347–351, 1999.

[15] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, 3rd ed. Pearson, 2010.

[16] Y. Yang and J. Pederson, "A comparative study on feature selection in text categorization," *Proceedings of the Fourteenth International Conference on Machine Learning*, vol. ICML'97, pp. 412–420, 1997.

[17] Keras - dense layer. [Online]. Available: https://keras.io/api/layers/core_layers/dense

[18] Deepai - bias vector. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/bias-vector

[19] Keras - dropout layer. [Online]. Available: https://keras.io/api/layers/regularization_layers/dropout

[20] Peltarion - categorical crossentropy. [Online]. Available: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy

[21] D. P. Kingma and J. Ba., "Adam: A method for stochastic optimization." *Technical Report 1412.6980*, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[22] Wikipedia - precision and recall. [Online]. Available: https://en.wikipedia.org/wiki/Precision_and_recall

Ringraziamenti

Un sentito ringraziamento al professore Maurizio Gabbrielli e alla ricercatrice Francesca Del Bonifro, che con la loro disponibilità e pazienza mi hanno supportato nell'ideazione e nella preparazione di questa tesi.

Infine un quantomeno banale, ma doveroso, ringraziamento alla mia famiglia e a tutti i miei amici, ogni traguardo è reale solo se viene condiviso.