

In Python si può passare anche una funzione come parametro ad un'altra semplicemente utilizzando il nome

In Python, una variabile esiste quando viene assegnato un valore

In Python 2 print era un comando, in Python 3 è diventata una funzione

```
a = 10  
print a #Python 2  
print(a) #Python 3
```

Noi useremo interprete Python almeno di 3.10 perché da questa sono stati introdotti alcuni costrutti importanti

Da Mac, dopo aver installato Python, bisognare runnare Install Certificates.command

Serve per scaricare dati da repository

Per rendere direttamente eseguibile (vale per tutti i linguaggi di scripting) uno script, basta inserire il path dove si trova l'interprete all'inizio del programma

Per vedere dove si trova da terminale: which python3

Se in uno script non c'è scritto niente (cioè non diciamo qual è interprete), la shell interpreta lo script come se stessa

if __name__ == '__main__' è un trucco per definire un main

Ciò è utile per evitare che i programmi vengano eseguiti direttamente quando vengono inclusi in altri programmi

JUPYTER NOTEBOOKS

Usiamo questo strumento come pane quotidiano

È un modo interessante per documentare i nostri programmi

È una sorta di pagina web interattiva dove si può mixare testo scritto con sintassi markdown o con formule latex con pezzi di codice che si può eseguire

Per usare i notebook bisogna installare jupyterlab con pip

Poi si lancia con il comando "jupyter lab"

Il kernel è l'interprete del notebook

Questo tool si può usare anche per R o Julia

Si possono definire celle di tipo markdown e celle di tipo codice

Partiamo con una cella markdown

```
#This is my first notebook
```

Premo play e mi formatta il fatto

è l'equivalente di h1 di html

La stessa cosa si può fare anche in visual studio code

Quando creiamo un nuovo file, possiamo creare un file jupyter notebook se c'è installato jupyter

L'idea del notebook è avere una light documentation

Un'altra alternativa per usare i notebook è Google Colab

Il vantaggio è che si può usare la potenza computazionale di google

Quindi è bello fare i progetti come notebook

Vediamo la sintassi dell'artefatto per far considerare due righe come se fossero nella stessa

```
x = 10
y = 20
if x > 5 and /
    y > 20
print('yes')
```

ASSIGNMENT

Quando si assegnano tipi primitivi, si assegna alla variabile un valore

Invece, con tutto ciò che non è un tipo primitivo, assegnamo una reference alla variabile (esempio: le liste)

Soltamente il camel case si utilizza solo per i nomi delle classi

Il doppio _ serve per rendere un attributo di una classe privato

Si possono fare assegnazioni multiple

```
x, y = 2, 3
```

Però è brutto

Tuttavia, è interessante però per scambiare due variabili

```
x, y = y, x
```

L'operatore di slicing : crea una nuova lista, cioè copia determinati elementi dalla lista di partenza in una nuova

```
mylistNew = mylist #copia il puntatore
mylistNew2 = mylist[:] #crea nuova lista
```

In Python non è detto che tutte le righe di una matrice devono avere lo stesso numero di elementi

Il metodo extend aggiunge una lista in coda ad un'altra

La differenza con + è che extend non crea una nuova lista

L'istruzione sort si può eseguire passando come parametro una funzione

▼ Lezione 4 - 11/03/2024

DIZIONARI

Com'è fatto un dictionary in Python? È molto simile ad una Java map

La caratteristica dei dizionari è che le **chiavi** devono essere **immutabili**

La sintassi per la creazione è con { }

```
# Esempio
mydict = {'red' = 20, 'blue' = 30}
```

Per accedere al valore si utilizza la sintassi: nomeDizionario[chiave]

```
mydict['red'] # Output: 30
```

I dizionari **NON** sono **ordinati**

Il metodo *clear()* rimuove tutti gli elementi dal dictionary

Ci sono metodi per ottenere tutte le chiavi o tutti i valori sotto forma di **lista**, utile per scorrere (cicli)

Si può avere anche ottenere una lista di item, in particolare si ottiene una lista di tuple

In generale, è buona pratica **testare** se una **chiave** è **presente prima** di fare un accesso al dizionario

Operazioni sulle stringhe:

- **upper**
- **lower**
- **strip**: quando apriamo un file e leggiamo le righe dal file, queste righe contengono i caratteri di ritorno a capo, se vado a stampare queste righe e ci metto un altro ritorno a capo, si portano il ritorno a capo appresso; tra l'altro il carattere di andata a capo è cambia a seconda dei sistemi operativi
- **join**

- **split**

L'istruzione **format()** è molto comoda

```
# Esempio
first_name = 'Francesco'
last_name = 'Iuorio'

s = "Firstname is {} and lastname is {}".format(nome, cognome)
```

Quando si vogliono aggiungere ad una stringa variabili che sono di un altro tipo, se non mettiamo **str()**, Python si arrabbia

ESPRESSIONI BOOLEANE

C'è un operatore **is()**, che ritorna True se due variabili fanno riferimento allo stesso oggetto

Operatore ternario in Python:

```
x = 10 if len(a) == 3 else 20
```

ciò è l'equivalente:

```
a.length == 3 ? x = 10 : x = 20
```

Per l'else-if esiste proprio il costrutto **elif**

Da Python 3.10 è stato introdotto il costrutto **match case**, simile allo switch case

Match specifica l'espressione da valutare

A differenza di Java, in Python il **break** alla fine di un blocco **case** **NON** è necessario, dato che se un case viene matchato viene eseguito solo quello

case_ è equivalente al Java default

```
sample = True
match sample:
    case (True|False):
        print("It is a boolean value")
    case _ :
        print("Not a boolean value")
```

In Python come valori dei case si può mettere di tutto, anche delle stringhe

Si potrebbe anche dichiarare una lista e mettere nel case un sottoinsieme della lista

In Python c'è l'istruzione **pass** per indicare che non deve fare niente. Viene utilizzata spesso quando si lavora con blocchi di if ma ancora non si è definito il comportamento per quel branch

Il ciclo for in Python è un po' diverso rispetto ad altri linguaggi perché esiste solo nella forma **for-each** (simile al for generalizzato di Java)

In generale, itera su una collezione

Un item può essere qualsiasi cosa

Ad **esempio** può essere una tupla:

```
for (x, y) in [("a", 1), ("b", 2), ("c", 3)]:
    print('key is ', x, "value is ", y)
```

Range ritorna una lista che parte dal primo valore specificato e si ferma al valore precedente all'altro parametro

È possibile anche utilizzare range indicando un passo

```
range(0, 10, 2)
```

Per stampare posizione corrente e elemento di una lista, la cosa più banale che si può fare è:

```
mylist = [1,2,3]
for x in range(len(mylist)) #crea range che va da 0 a 2
    print(x, mylist[x])
```

Oppure si può utilizzare la funzione **enumerate**

Un oggetto **enumeration** contiene tante tuple e ognuna contiene posizione e elemento

```
for (x,y) in enumerate(mylist)
    print(x, y)
```

LIST COMPREHENSION

Si tratta di un meccanismo che permette di generare liste con una sola espressione molto sintetica

Sintassi: **[expression for name in list]**

FILTERED LIST

Alle **list comprehension** si può aggiungere una condizione

Sintassi: **[expression for name in list filtro]**

La parte di iterazione funziona come prima, ora in più viene testata la condizione, l'espressione viene valutata e il risultato di questa viene aggiunto alla lista solo se la condizione è verificata

È possibile anche fare list comprehension innestata

```
# List comprehension innestata  
mylist = [0 for i in [for j in range(10)]]
```

Questa roba è utile quando si hanno molte strutture dati

PYTHON FUNCTIONS

Tutte le funzioni Python hanno un valore di ritorno

Una funzione che non ritorna niente, in realtà **ritorna un oggetto di tipo None**,

None è una costante speciale in Python, simile al null java anche se logicamente è più simile al false

Se qualcosa è None non viene stampata dall'interprete

Si può però controllare se è None

```
x is None
```

In Python NON esiste il concetto di overloading

Le **funzioni** sono dette **first class objects**, cioè le funzioni sono oggetti, infatti si può passare una funzione come parametro ad un'altra

I parametri si possono passare sia per valore che per reference

Python permette anche un passaggio di parametri per **keyword**

Infatti è possibile chiamare la funzione sia in maniera posizionale, sia con le coppie nomeparametro-valore, sia in maniera ibrida

I vantaggi di questa possibilità sono che non dobbiamo ricordare l'ordine dei parametri e che è permesso definire valori di default

LAMBDA FUNCTION

La lambda permette di definire una funzione **on-the-fly**

Ad esempio, si potrebbe passare il lambda ad una funzione che filtra una lista
filter() è una funzione che data una lista, applica una funzione filtro che se la condizione è vera restituisce l'elemento, altrimenti no
filter() restituisce un oggetto di tipo **filter**

La funzione **map** applica una funzione su tutti gli elementi di una lista

Un argomento preceduto da * impacchetta elementi in una **lista (*args)**

Un argomento preceduto da ** impacchetta elementi in un **dizionario (**kwargs)**

Il concetto di **packing** è utile anche per fare funzioni con numero di parametri variabili

L'operazione duale è l'**unpacking**

Esempio: se ho una funzione convenzionale che prende 3 parametri e questi 3 parametri li ho in una lista, allora posso chiamare la funzione passandogli ***nomeLista**

▼ Lezione 5 - 14/03/2024

Vediamo come utilizzare Python in modalità OO

A differenza di Java dove esistono i tipi nativi, in Python tutto è un oggetto

Inoltre non andremo a definire esplicitamente attributi (variabili d'istanza), ma li creiamo implicitamente nel costruttore (stile js)

Per dichiarare un metodo, la stessa sintassi delle funzioni (`def`)

La differenza tra un metodo e una funzione è che il primo ha come parametro `self` (puntatore all'oggetto)

Il metodo `__init__` è il **costruttore**

Anche in questo linguaggio non c'è bisogno di un distruttore a differenza di C++

Si può accedere ad un metodo o attributo anche **dinamicamente**

Per farlo, si può passare al metodo `getattr` il nome del metodo o attributo a cui si vuole accedere

Esiste anche il metodo `hasattr` che restituisce True se un metodo ha un attributo o meno

DATA ATTRIBUTES VS CLASS ATTRIBUTE

Con **data attributes** si intendono l'equivalente delle **variabili d'istanza** di Java

Con **class attributes** invece si intendono le variabili d'istanza **statiche**

I **data attribute** vengono inizializzati con il metodo `__init__` o, in generale, con la sintassi `self.nomeAttributo`

Invece per inizializzare un **class attribute** devo anteporre al nome dell'attributo la **keyword** `__class__`

```
class sample:  
    x = 23  
  
    def increment(self):  
        self.__class__.x += 1
```

INHERITANCE

Python permette di fare più cose di quanto permette Java

Infatti qui è permessa l'**ereditarietà multipla**

NON c'è una parola chiave come extend, ma il **nome** della **classe** da **ereditare** viene **passato** come **parametro** alla **sottoclasse**

```
class Studente:  
    def __init__(self,n,a)  
        self.num = n  
        self.attr = a  
  
class Cs_student(Student):
```

```
def __init__(self, n, a, s):
    Student.__init__(self, n, a) # equivalente del super in Java
    self.section_num = s
```

In Python **NON** è permesso l'**overloading**, ma **è permesso l'override**

METODI SPECIALI:

- **__init__**: è il costruttore
- **__cmp__** : equals in Java
- **__len__**
- **__copy__**
- **__str__** : toString di Java

Una cosa divertente è che è possibile **ridfinire il comportamento** degli **operatori**

Esempio: si può utilizzare per implementare l'aritmetica dei numeri complessi o per le matrici

Se **eq** NON è implementato, un confronto tra due oggetti restituisce True se puntano allo stesso oggetto

Non esistono **qualifier** di **visibilità**

Tuttavia con **due _ davanti un attributo**, questo diventa **privato**

Mentre **NON** esiste in nessun modo il concetto di **protected**

IMPORTING MODULES

Vediamo come gestire i package di Python

La parola chiave è **import** per utilizzare un libreria (**modulo** più corretto)

Python **NON** cerca i moduli **solo** nella **directory corrente**

Le librerie si possono installare sia **a livello globale** sia in un **virtual environment**

Python **cerca** le librerie all'interno di un set di percorsi chiamato **sys.path**

`sys.path` è accessibile da una libreria chiamata `sys`

VIRTUAL ENVIRONMENT

In generale, quando si distribuisce un progetto, il codice preso così com'è non funzionerà perché nei progetti vengono importate alcune librerie

Un **virtual environment** è un ambiente confinato all'interno del quale, per un dato progetto, si installa una versione di interprete Python e alcune librerie

Per fortuna, il sistema operativo è intelligente e crea link simbolici, non installa ogni volta i pacchetti

Ciò permette anche di avere una lista di **quali librerie (e quale versione)** sono state utilizzate per il progetto

Dopo aver creato il **virtual environment** bisogna **attivarlo** con il comando:

source .venv/bin/activate

Se ho installato librerie globalmente, non le vedo nel virtual environment

pip freeze restituisce la lista di librerie installate utilizzando la sintassi dei **requirement**

Prima di distribuire lo script, si usa pip freeze per indirizzare il contenuto in requirements.txt:

pip freeze > requirement.txt

Un package può contenere più file

Importare un file come modulo è banale

Invece se volessi importare un'intera directory, allora questa deve contenere un file vuoto chiamato `__init__.py`

Alcuni moduli che useremo molto sono `sys` e `os`, i quali sono moduli predefiniti

Il modulo `sys` ha funzioni e variabili di uso ricorrente, ad esempio:

- `sys.path`
- `sys.argv` (è una lista)
- alcune variabili sono `stdin`, `stdout` e `stderr`

Per ridirigere l'output sullo `stdout` e `stderr` da riga di comando:

`python prova.py >out 2>errors`

FILE I/O

`readLines` legge tutte le linee del file

con `readLine` lo svantaggio è che carichiamo tutto il file in memoria

Conviene usare il puntatore al file

Per scrivere sul file si può usare sia `write` che `print`

`read(1)` legge un carattere alla volta

Scrivendo “`python prova.py <errors`” possiamo utilizzare `errors` come canale di input

▼ Lezione 6 - 18/03/2024

ECCEZIONI IN PYTHON

La gestione delle eccezioni non è molto diversa da Java

Le eccezioni sono oggetti Python

Le keywords per interagire con le eccezioni sono:

- `try`

- `except`
- `finally`

Nel blocco `except` il nome dell'eccezione è opzionale, infatti se non viene esplicitato viene catturata un eccezioni **generica**

Esiste anche l'opportunità di utilizzare la clausola `else`, la quale viene eseguita solo se NON si verificano eccezioni

`finally` invece si esegue a prescindere

La clausola `raise` è l'equivalente del `throw` in Java

LIBRERIE PYTHON PER ML

Le librerie general purpose più utilizzate per **ML** e **IR** sono:

- **numpy**: permette di gestire strutture dati, come matrici e vettori, in maniera più efficiente; permette di trattare matrici come **matrici sparse**; permette di eseguire operazioni di algebra lineare, trasformate di Fourier
- **scipy**: implementa algoritmi scientifici
- **matplotlib**: serve per fare grafici
- **pandas**: la più importante per gestire dataframe
- **nltk**: per il natural language processing

Matrice sparsa: matrice i cui valori sono quasi tutti uguali a zero

PANDAS

Permette di gestire i **dataframe**, cioè la tipica struttura dati utilizzata in statistica e ML

Dataframe: dataset organizzato in righe (omogenee) e colonne (eterogenee)

I dataframe sono importanti in ML perché ogni **riga** identifica un **campione**, mentre ogni **colonna** è una **feature**

Soltamente i dataframe sono file comma separate value

Per creare dataframe, in pandas esiste l'operatore `DataFrame` che prende come parametro i valori da aggiungere al dataframe

Un dataframe può essere creato sia a partire da una lista sia da un dictionary

Se all'interno di un dictionary avessimo **liste** con **cardinalità minore** rispetto ad altre, allora le prime verrebbero **iterate** fino a raggiungere il riempimento del dataset

La funzione `describe()` calcola degli **indici statistici** per ogni colonna

È possibile fare un sort del dataframe e fare anche operazioni di **selezione** su **condizione**

Pandas usa `np.nan` per rappresentare **dati mancanti**

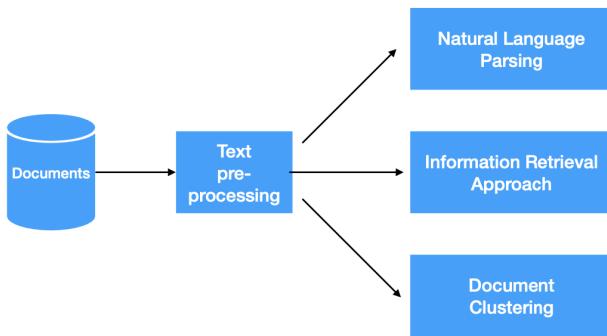
Poi ci sono funzioni che permettono di **eliminare** righe, **riempire** dove ci sono valori mancanti e/o di **controllare** dove sono presenti o meno dei valori

La funzione `apply()` permette di applicare un certo operatore su tutti gli elementi del dataframe. Accetta come parametro anche una **lambda**

TEXT PREPROCESSING

Vedremo come funziona il NLP, quali sono i primi passi da compiere e le **espressioni regolari**

Typical NLP approach



NL PARSING

Ciò che si fa è creare un **albero sintattico** della frase, quindi capire chi è il soggetto, qual è il verbo, ecc.

INFORMATION RETRIEVAL

È il processo con il quale partendo da una necessità di avere delle **informazioni**, queste vengono chieste ad un **motore** di **ricerca** che ritorna dei risultati

Anche i LLM potrebbero essere considerati come un motore di ricerca, anche se differiscono dal modo in cui restituiscono i dati

Un **modello neurale** puro **NON** ha un **database** dal quale recuperare informazioni, bensì prende la nostra query e esegue una trasformazione dell'input in un output

Esistono anche modelli ibridi

PROBLEMA DELL'INFORMATION RETRIEVAL

L'obiettivo è trovare documenti rilevanti per l'informazione richiesta da una grande quantità di documenti

In generale, formulo una **query** e il sistema di IR **recupera** informazioni da una **collezione di documenti** e **restituisce** una **lista**

Se mi accorgessi che ci sono informazioni **errate** nel database, allora si può tranquillamente **modificare il database**

Invece, per un **modello neurale** dato che non c'è un database dal quale recuperare informazioni, è molto più complicato correggere degli errori in quanto si tratterebbe di **modificare** dei **coefficienti**, cosa per niente banale

Tipicamente quando si vuole fare **clustering** di **documenti** o **IR**, bisogna fare un processo di **text processing** importante

Quando si vuole utilizzare l'approccio **NL parsing** **NON** si fa un grande **preprocessing** perché un **NL parser** ha bisogno di **tutta la frase** (comprese congiunzioni, plurali)

Invece, per un sistema di **IR**, questi elementi devono essere eliminati perché li mandano un po' in crisi, anche se al giorno d'oggi i problemi sono stati abbastanza superati

DIETRO UN SISTEMA DI IR

Si parte da dei documenti e da una query, bisogna **trasformare il documento** e la **query** tramite un operazione di **indicizzazione**

Tipicamente ogni **documento** diventa un **vettore**

Questo processo richiede una serie di passi:

1. **estrazione dei termini**: se ho una frase, devo estrarre tutte le parole dalla frase; si possono escludere caratteri speciali e utilizzarli come separatori, anche se non è così banale.

Esempio "open-source": tipicamente sono tentato a eliminare il "-", però ciò mi porterebbe ad avere due parole differenti che hanno una semantica diversa rispetto a quella che avrebbe se fosse una sola. Un altro **eSEMPIO** è Ph.D.

Spesso i numeri non vengono eliminati se hanno una semantica, **esempio**: anno in un documento storico

- **eliminazione di stop word**: si eliminano quelle **parole** che NON danno **informazioni specifiche** (articoli, preposizioni)
- **stemming** o **lemmatization**: si intende **riportare** le **parole** ad una **radice comune**, ciò si può fare in due modi: con lo **stemming** si applicano **algoritmi**, si applicano regole che funzionano con tutte le parole; invece la **lemmatization** usa un **database lessicale** dove ci sono tutte le parole della lingua e le principali relazioni

Lo **stemming** è più **leggero** a livello **computazionale**, tuttavia **NON** sempre restituisce una **parola di senso compiuto**, ma una **parola troncata**

Esempi dove lo **stemming** funziona bene:

- cars → car
- computer → compute

Lo **stemming** **NON** funziona **bene** con **lingue complesse** o con **verbi irregolari** dell'inglese

Esempio: went → go

Questo esempio si risolve facilmente con la **lemmatization** dato che prevede un database che contiene tutte le forme verbali di go

DOCUMENT CLUSTERING

Per fare clustering si utilizzano le stesse tecniche utilizzate nell'IR

REGULAR EXPRESSIONS

È il primo strumento da conoscere per fare text processing

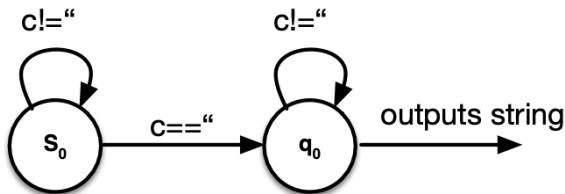
Si possono utilizzare anche per definire una **modalità di scripting** di un documento

Dietro al funzionamento di una **espressione regolare** c'è un **automa a stati**

Il **lexer**, la componente che identifica espressioni in un linguaggio, funziona con un **automa a stati**

Immaginiamo di voler scrivere un **espressione regolare** che estrae tutte le **costanti stringa** da un documento preso in input

Ipotesi: se ci sono " allora si tratta di una stringa



Mi metto nello stato iniziale e **leggo** finché il carattere è diverso da "", a questo punto **finisco** di leggere e passo nello stato di **acquisizione stringa** finche non trovo un'altra " e **stampo** la stringa

Questo è un **eSEMPIO** di estrazione stringa da un documento.

Possiamo usare le espressioni regolari per estrarre stringhe, questo strumento permette di ridurre di molto le righe di codice di un programma

Un tool per testare espressioni regolari:

regex101: build, test, and debug regex

Regular expression tester with syntax highlighting, explanation, cheat sheet for PHP/PCRE, Python, GO, JavaScript, Java, C#/.NET, Rust.

R <https://regex101.com>

Does `\bR\b \eg\b[Ee]\xp\b . confuse - you - too\b\? . Visit \.(regex101\.com)`

syntax highlighting | regex breakdown | code generator | cheat sheet
regex debugger | community patterns | unit tests | regex quiz

DISJUNCTION

Una disgiunzione è un qualcosa che vado a racchiudere all'**interno** di [] del **pattern** dell'**espressione regolare**

I pattern che si trovano all'interno delle [] sono quelli che possono comparire in una stringa

Esempio: [wW]oodchunck

Ciò significa che in prima posizione **dove apparire w o W** affinché avvenga il match

Esistono anche **shortcut**, utilizzate per i **range**:

- [A-Z]: matcha una qualsiasi lettera maiuscola
- [a-z]: matcha una qualsiasi lettera minuscola
- [0-9]: matcha un qualsiasi numero intero

All'interno di una disgiunzione si può inserire una **negazione**, cioè in un punto non deve esserci quello tra []

La negazione va messa all'inizio

Esempio: [^A-Z] indica che in un punto non deve esserci una lettera maiuscola

Si può anche utilizzare l'operatore **or** |, il quale indica match se è presente un'espressione tra le opzioni

CARATTERI SPECIALI NELLE REGEX

- ?: indica che il carattere precedente è **opzionale**
- * : indica la presenza di 0 o più ripetizioni del carattere precedente
- + : indica la presenza di 1 o più ripetizioni del carattere precedente
- . : indica che in quel punto può esserci un carattere **qualsiasi**
- ^ : quest'operatore posto all'inizio di una regex, indica che la stringa deve iniziare con quel pattern
- \$: è l'operatore duale a ^, infatti indica che la stringa deve terminare con il carattere precedente, quindi considero la fine di una stringa

Con \.\$ indichiamo una regex che matcha una stringa se questa **termina** con il **punto**

Invece, se **NON** avessimo messo \., ma solo ., allora per far avvenire il match, sarebbe bastato un carattere qualsiasi

Altre shortcut, le quali ci evitano l'utilizzo delle disgiunzioni:

- \d: indica che in quel punto deve esserci una **cifra**

- **\s**: indica che in quel punto deve esserci un carattere di **spazio** (matcha anche ritorno a capo e tab)
- **\w**: matcha **caratteri alfanumerici**
- **\b**: matcha i **confini** di una **parola**

Questi 4 pattern hanno un loro duale, in particolare la sintassi per ottenere il **negato** di questi è la stessa con la differenza che si utilizza la lettere **maiuscola**

\b mi permette di dire che c'è un separatore di parole nel pattern

Esempio:

- `\bcar\b` matches “car”, “.car”, “car go”, but not “caret”
- `car\b` matches “car”, “car.”, “supercar” but not “caret”
- `\Bcar` matches “supercar” but not “this is my car”

Una delle prime cose che fa un linguaggio di programmazione è la **tokenizzazione** per trovare gli **identificatori**

La regex utilizzata da Python è:

```
/ \b[a-zA-Z_] [\w_]* \b /
```

Da ciò si capisce che in Python non sono ammessi identificatori che iniziano con un numero

RANGES

{x,y}: indica che un pattern si può ripetere **da x a y volte**

{x}: indica che un pattern si può ripetere **esattamente x volte**

{x,}: indica che un pattern si può ripetere **almeno x volte**

Il pattern da ripetere va messo tra **()**

Esempio:

(go) {2}: matcha "gogo" e non "go"

go {2} invece riferisce il 2 solo alla "o" quindi matcherebbe "go" e non "gogo"

▼ Lezione 7 - 21/03/2024

Abbiamo visto che in generale, data una stessa parola, si possono scrivere **regex** che funzionano in maniera diversa

Esempio: vogliamo trovare tutte le istanze di "the" in un testo

Allora potrei scrivere una regex che

NON considera che l'iniziale può essere anche maiuscola e che matcha anche un "the" come sottostringa

Quindi, quando si matcha qualcosa nel testo si possono avere:

- **falsi positivi** (detti **errori di primo tipo**) quando **matchiamo** qualcosa che **NON volevamo**
- **falsi negativi** (detti **errori di secondo tipo**) quando l'algoritmo di ricerca **NON** è in grado di **restituire** alcun **elemento**

Quando realizziamo un algoritmo di **classificazione e ricerca**, gli obiettivi che vogliamo raggiungere sono di avere un algoritmo:

- **accurato**, cioè che **minimizzi i falsi positivi**
- che ci restituisca **più risultati possibili**, cioè che **minimizzi i falsi negativi**

Questi sono due obiettivi spesso contrastanti, infatti di solito si penalizza un aspetto rispetto all'altro

SOSTITUZIONI

Un'altra applicazione delle **regex** è quella di **trasformare il testo** e questo è possibile con le **espressioni di sostituzione**

L'idea è di avere un **espressione regolare** che se **matchata**, allora venga **rimpiazzata** con un determinato **pattern**

Esempio perl: s/colour/color

Se si matcha colour, questo viene sostituito con color

Supponiamo di voler fare una piccola trasformazione del testo

Ogni volta che troviamo un numero, lo vogliamo racchiudere in <>

Esempio: **the 35 boxes → the <35> boxes**

Per farlo specifichiamo ciò che vogliamo matchare e inseriamo il pattern in delle **parentesi tonde**, le quali servono sia per dare una precedenza sia per **identificare** un **gruppo** da **estrarre**

In generale, se racchiudiamo qualcosa tra **parentesi tonde**, il **pattern** verrà identificato come un gruppo che avrà una **identificativo numerico**

Infatti, per far riferimento al gruppo 1 si può usare l'operatore \1

Esempio: s/ ([0-9]+) / \1 oppure s/ (\d+) / <\1>/

[0-9]+ == \d+

Gli operatori \1\2 servono per riferire **gruppi di espressioni** racchiusi tra ()

Per **evitare** che un gruppo venga **riferito** con **\numero**, quindi se voglio un gruppo che non voglio mai riferire, si può **indicare** di **NON contare** il gruppo

Per farlo, si aggiunge ?: prima dell'espressione che **NON** si vuole contare

Esempio: (?:some|a few)

LOOKAHEAD ASSERTIONS

Ci permettono tramite un **operatore di definire** un **match** o **NON match** all'interno di un gruppo e far sì che l'analizzatore di testo **NON** vada avanti a matchare

Supponiamo di voler risolvere il seguente problema: vogliamo matchare una stringa che inizi con caratteri alfabetici a patto che non inizi con la parola "Volcano"

Come posso fare questa cosa?

Scriviamo un'espressione del tipo: /^(?!Volcano) [A-Za-z]+/

In generale:

- **(?= pattern)** è vera se il pattern è **matchato**
- **(?! pattern)** è vera se li pattern **NON matcha**

SIMPLE APPLICATION: ELIZA

È un primo esempio di chatbot

Il funzionamento era molto semplice, infatti matchava una stringa fatta in un certo modo e facevano una specie di search and replace

Il match veniva fatto proprio con delle espressioni regolari

REGEX IN PYTHON

Per poterle utilizzare dobbiamo utilizzare il package **re**

Le prime due funzioni che vediamo sono:

- **search**: cerca un pattern all'**interno** di una stringa
- **match**: cerca il pattern all'**interno** di una stringa

Il funzionamento quindi è abbastanza simile

Esempio: supponiamo di voler scrivere un'espressione che matchi un indirizzo di posta

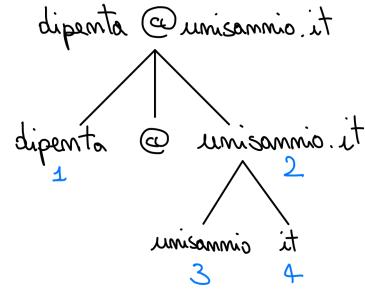
L'espressione regolare sarà: **\w+@\(\w\.)\(\com|\org|\net|\edu|\it)**

Con l'operatore **group** si può accedere ai **gruppi** di **espressioni regolari**

Per **decidere** il numero di gruppo, viene **costruito** una specie di **albero**

I numeri vengono dati in base all'**ordine** di **attraversamento** dell'albero, viene attraversato in ampiezza

ESEMPIO: dipenta @ unisannio.it



Si può anche assegnare un nome simbolico ad un gruppo, si fa con **P<>**

Esempio: (**P<parola> \w+**), accediamo al gruppo con `group('parola')`

PRE MATCH E POST MATCH

Spesso potrebbe interessare ciò che si trova **prima** o **dopo** di quello che si **matcha**

Per accedere al **prematch** si può fare uno slice della stringa, cioè facendo:

```
r1.string[:r1.start()]
```

Invece, per accedere al **postmatch**: `r1.string[r1.end():]`

Perl ha queste funzioni già implementate nativamente

Altri operatori che utilizzeremo sono:

- `split()`, lo conosciamo già sulle stringhe, solo che sulle regex usa come **separatore** un **pattern** e non un carattere
- `sub()`, è la funzione di **sostituzione**, permette di eseguire dei **search and replace**
- `.findall()`, funziona in maniera leggermente diversa da search, **restituisce una lista contenente tutti i match** in una stringa

Esempio:

```
re.sub('(blue|white|red)', 'black', 'blue sock and red shoes')
```

Con quest'istruzione, andiamo a sostituire

blue e red con black

È possibile anche **compilare** una **regex**

Compilare è utile perché permette che il **matching** avviene più **velocemente**, più efficiente

Inoltre se si verifica un errore del matching, lo si scopre subito

MATCHING OPTIONS

Le **opzioni di pattern matching** sono molto utili

Le funzioni `search`, `sub` accettano come **parametro l'opzione di matching**

Una cosa utile potrebbe essere rendere il matching case **unsensitive**, per farlo si passa come terzo parametro delle funzioni di **re** con l'espressione `re.IGNORECASE`

Un'altra cosa interessante è il matching **multilinea**

Esempi:

- `re.search("bug", b, re.IGNORECASE)` b is matches if it is equal to "Bug", "BUG", "bug" etc.
- `re.findall("^max", "max is going\nmax is coming", re.MULTILINE)`
['max', 'max']
- `re.findall("^max", "max is going\nmax is coming")`
['max']

RICERCA GREEDY VS LAZY

Ci serve per far capire come funzionano espressioni regolari quando vanno a matchare caratteri speciali

In **match greedy** si cerca di matchare la sequenza più lunga possibile, quindi se ad esempio volessimo matchare del testo che si trova tra "", verrebbero ignorate, qualora ce ne fossero, le virgolette in mezzo

Se invece volessimo fermarci alla prima virgoletta possiamo utilizzare il seguente codice:

```
import re
>>> mystr = 'My text says "I am very happy" and "I do not want to go
>>> mystr
'My text says "I am very happy" and "I do not want to go to school"'
>>> re.search("\\"(.+)\\"", mystr)
<re.Match object; span=(13, 66), match='I am very happy' and "I do
>>> r1 = re.search("\\"(.+)\\"", mystr)
>>> r1
<re.Match object; span=(13, 66), match='I am very happy' and "I do
>>> r1.group()
'"I am very happy" and "I do not want to go to school"'
>>> r1 = re.search("\\"([^\"]+)\\"", mystr)
>>> r1.group()
'"I am very happy"'
>>> myhtml = "<body><h1> this is my text </h1> </body>"
>>> r2 = re.search("<(.+)>", myhtml)
>>> r2.group
<built-in method group of re.Match object at 0x102b45b40>
>>> r2.group()
'<body><h1> this is my text </h1> </body>'
>>> r2 = re.search("<([^\>]+)>", myhtml)
>>> r2.group()
'<body>'
```

Un altro modo per fare questa cosa è un match **pigro (lazy)**

Si può fare mettendo il ? all'interno del gruppo, cioè `\\"(.+?)\\"`

TEXT TOKENIZATION

Per **analizzare** un documento, dobbiamo estrarre frasi e di conseguenza le parole

In generale, quando applichiamo **NLP** dobbiamo:

- **suddividere** un documento in **frasi** (il metodo più semplice potrebbe essere quello di fermarsi dove troviamo un **punto**, ma avremmo problemi se nel testo ci fosse qualcosa come "Mr. Smith" oppure un email)
- **normalizzare il formato** delle **parole** (per convenzione lavoriamo di solito con parole in **minuscolo**)
- **suddividere la frase in parole**

Come dividere una frase in parole?

Potremmo utilizzare gli **spazi** come riferimento per suddividere una frase in parole

Da terminale **UNIX** con il comando **tr** si può **sostituire** un **pattern** con un altro all'interno di un documento

tr 'A-Z' 'a-z': trasforma uppercase in lower case

SENTENCE SPLITTING

La prima cosa che si può fare è prendere un **documento**, **leggerlo** e assegnarlo ad una stringa, dopodiché usare la funzione **split** per andarla a dividere

Esempio: split se c'è uno spazio dopo un segno di punteggiatura

```
>>> import re
>>> text = 'Mr. Smith is buying flowers, plants, etc. before going to work'
>>> sentences=re.split(r"[\.\!?\]\s+", text)
>>> sentences
['Mr', 'Smith is buying flowers, plants, etc', 'before going to work']
```

Però **NON** è perfetto in quanto "Mr. Smith" viene considerato come due frasi

Un **trucco** potrebbe essere **sostituire tutte le occorrenze** di **Mr.** con **Mr**

Inoltre, potremmo assumere che parole come **mr.**, **dr.**, **prof.**, **i.e.**, **e.g.** **NON** compaiono solo alla fine della frase

```

import re

prefix="mr|prof|dr|mrs|eng|i.e|e.g"
s="Mr. Smith is buying things e.g., flowers, plants, etc. before going to work with Prof. John"

s2=re.sub("(?i:" + prefix + ")\.", r"\1", s, flags=re.IGNORECASE)
sentences=re.split(r"\.|\!?", s2)

```

Potremmo avere il problema **etc.** perché spesso compare alla **fine** della frase e quindi se **rimuovessimo** il punto, rischieremmo di perderci la **fine** della frase

```

import re

prefix="mr|prof|dr|mrs|eng|i.e|e.g|etc"
abbreviation="etc"

s="Mr. Smith is buying things e.g., flowers, plants, etc. before going to work with
Prof. John"

s2=re.sub(r"\b"+r"(?i:"+abbreviation+r")\.(?s+[\n\r],){0};:-]*\s*[A-Z])", r"\1 .\2", s)
s3=re.sub("(?i:" + prefix + ")\.", r"\1", s2, flags=re.IGNORECASE)

sentences=re.split(r"\.|\!?", s3)

```

Quindi la **soluzione** per **etc** è quella di creare una **regex** che **sostituisca** con la variante **senza punto solo se si rispettano** alcune **condizioni** tra cui che la **prima lettera dopo** sia una **maiuscola**.

I LLM tutti questi problemi li hanno già risolti

▼ Lezione 8 - 25/03/2024

Abbiamo visto come si potrebbero utilizzare regex per **pre-processare** del testo

Un tipico problema si ha con parole tipo **etc.**

Questo tipo di parole possono comparire sia in mezzo che alla fine di una frase

Una **soluzione** può essere quella di creare una lista di **prefissi** e di **abbreviazioni**

La differenza è che le **abbreviazioni** sono quelle **parole puntate** che **possono anche comparire alla fine** della frase

IMPLEMENTAZIONE IN NLTK

La libreria `nltk` ha una funzione `sent_tokenize` che ci aiuta a realizzare questo processo

Con un'altra funzione, `PunktSentenceTokenizer`, si può creare una **lista** con **tutte** le **abbreviazioni** da utilizzare per **gestire** la mia **frase**

```
from nltk.tokenize.punkt import PunktSentenceTokenizer, PunktParameters

punkt_param = PunktParameters()
abbreviation = ['mr', 'u.s.a', 'fig', 'etc', 'i.e', 'e.g']
punkt_param.abbrev_types = set(abbreviation)
tokenizer = PunktSentenceTokenizer(punkt_param)

s="Mr. Smith is buying flowers, plants, etc. before going to work"
sentences=tokenizer.tokenize(s)

for sentence in sentences:
    print(sentence)
```

```
Mr. Smith is buying flowers, plants, etc. before going to work
```

Però, in questo modo, **NON** si gestisce l'**etc** alla **fine** della frase

La funzione `tokenize` è **customizzabile**

Per gestire l'**etc finale**, dobbiamo inizializzare la lista delle **abbreviazioni**, poi vediamo se le **postfix abbreviation** compaiono alla fine e nel caso aggiungiamo lo spazio

Andiamo a gestire il tutto con una funzione di **sostituzione**, in particolare usiamo una **regex**

```

from nltk.tokenize.punkt import PunktSentenceTokenizer, PunktParameters
import re

punkt_param = PunktParameters()
abbreviation = ['mr', 'u.s.a', 'fig', 'etc', 'i.e', 'e.g']
postfix_abbr="etc"
punkt_param.abbrev_types = set(abbreviation)
tokenizer = PunktSentenceTokenizer(punkt_param)

s="Mr. Smith is buying flowers, plants etc. today, he's not feeling well"

s2=re.sub(r"\b"+r"("+postfix_abbr+r")\.\.(\s+\[\],()\{};:-]*\s*[A-Z])",r"\1 .\2",s)
sentences=tokenizer.tokenize(s2)

```

TOKENIZZAZIONE

Vediamo come **splittere** una **frase in parole**

Il modo più banale è usare gli **spazi** e lo facciamo con la funzione `split` del package `re`

In realtà, `split` **NON** è la funzione migliore perché tutti i **caratteri speciali** vengono **"attaccati"** alle parole

Ciò che ci conviene fare è una **lista** contenente i **caratteri con cui vogliamo fare lo split**

La prima cosa da **eliminare** sono le **parentesi** perché di solito sono attaccate alle parole

Dopodiché vanno **eliminati alcuni segni di interpunkzione** solo **se sono seguiti da spazi**

Esempio: un ":" che si trova all'interno di un **orario NON** va eliminato

Ora proviamo una funzione messa a disposizione dalla libreria **nltk** in modo da **NON usare regex esplicitamente**

La funzione è: `word_tokenizer`

Il problema di questa funzione è che **considera** dei **caratteri speciali** come **parole**

Inoltre, dopo aver **tokenizzato** dovremmo fare una **sostituzione** per le **forme contratte**

Esempio: 'm → am

Quindi i caratteri speciali vanno rimossi dopo

Vediamo dei **tokenizzatori** da utilizzare quando si usano librerie di deep learning

Questi operano osservando montagne di testo e **cercano di capire** quali sono le **singole parole** all'interno di un testo, **NON** usano regex

Per usare questi tokenizzatori servono una serie di librerie:

- `tensorflow` (la utilizzeremo spesso)
- `transformers`
- `sentencepiece` (libreria che permette di creare tokenizzatori)

Queste librerie permettono di usare **tokenizzatori pre addestrati**

`transformers` mette a disposizione `BertTokenizer`

In Python, dopo aver importato la classe, bisogna scaricare il **tokenizer specificandone il nome**

Il **tokenizzatore** è un **modello** di machine learning

Proviamo con `bert-base-uncased`

Dato che è uncased **converte** in **minuscolo** tutte le parole

Però anche questo considera delle **virgole** come parole, quindi bisogna gestirle

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

sentence = "I just bought a new GPU"
tokenizer.tokenize(sentence)

['i', 'just', 'bought', 'a', 'new', 'gp', '##u']
```

significa che è una parte di parola

GPU è stata splittata in due parole, dato che viene convertito tutto in **lower case**, gpu in minuscolo non esiste, per cui il tokenizer la splitta con parole che conosce

Tuttavia questo fatto può essere utile nel caso di parole come "creditcase" che viene splittata in "credit" e "case"

Ovviamente, esiste anche una versione **cased** del tokenizer, però va importata la classe `XLNetTokenizer` sempre da `transformers`

In generale, da ciò capiamo che:

numero di parole ≠ token

Questo accade proprio perché nei **token** portiamo **caratteri speciali** o addirittura **parti di parole**

WORD NORMALIZATION

Dopo aver tokenizzato le parole, bisogna **normalizzarle**

Dobbiamo capire come gestire coniugazioni dei verbi, ecc

Nella maggior parte dei casi, la soluzione è portiamo **tutto in minuscolo**

Però, in alcuni casi potrebbero portare due parole diverse

Esempio:

US → united states

us → noi

Quindi dovremmo portare tutto in minuscolo, **eccetto alcune parole** che sono in una lista speciale

Potremmo creare una **lista di acronimi** o, in generale, **di parole** che **NON** vogliamo convertire **in lower case**

STOP WORD REMOVAL

Questo procedimento è necessario solo con i modelli più semplici di IR

Le **stop word** sono **parole contenute** in una **particolare lista** (ce ne sono tante di liste di stop word per ogni lingua)

`nltk` facilita il tutto, è presente un package che contiene liste di **stop word** di varie lingue

È importante **identificare** le **stop word** perché si ha l'obiettivo di **indicizzare documenti** con **keyword**, cioè parole significative

In realtà modelli neurali, **NON** hanno bisogno della rimozione delle stop word

STEMMING E LEMMATIZATION

L'obiettivo è **riportare le parole ad una radice comune**, cioè se ci sono plurali o coniugazioni di verbi, si **evita** che **diverse forme** della **stessa parola** vengano **considerate** come **parole differenti** quando si costruiscono modelli

Lo **stemming** è un approccio **a regole**

La **lemmatizzazione** è basato su un **database morfologico**

STEMMING

Nasce soprattutto su lingue come l'inglese e non funziona benissimo con l'italiano

Ciò è dovuto al fatto che il **modo** con cui si possono **coniugare verbi** e come fare **plurali** è quasi sempre lo **stesso**

Cioè ci sono una **serie di regole** che **se la parola gode** di determinare **proprietà**, allora si possono **troncare** e **ridurre** ad una **radice comune**

Spesso si arriva anche a parole che **NON** esistono, dette **stem**, che sono la **radice comune**

Vediamo l'algoritmo più semplice, che non è quello che funziona meglio

Altri funzionano con stessa logica ma danno risultati migliori perché hanno regole più specifiche

Algoritmo di PORTER

Produce degli **stem**, i quali **NON** sono parole della lingua

Un problema è che spesso **produce** dei **troncamenti in eccesso** che fanno sì che si **uniscano parole** che **NON hanno niente in comune**

In generale, **tutti gli stemmer** non riconoscono parole di derivazione morfologica (go e went due parole diverse)

Come funziona quest'algoritmo? **Riconosce consonanti e vocali**

Indichiamo una consonante con **c** e una vocale con **v**

In questo modo, qualsiasi parola può essere ottenuta con diverse combinazioni, ad **esempio**:

- CVCV ... c
- CVCV ... v
- VCVC c
- VCVC ... v

Quindi, con questa rappresentazione potremmo avere parole di diverse combinazioni

Dopodiché si **conta quante volte** la coppia **VC** si **ripete** e si indica il **numero** con **m** (la notazione è **(vc){m}**)

m è **fondamentale** per applicare l'algoritmo

- **m = 0**: indica che la parola **inizia** con un **gruppo** di **consonanti o vocali**, ma **NON** è **seguita** da una **coppia vc**
- **m = 1**: indica che la parola ha **una ripetizione** della **coppia vc**
- **m = 2**: indica che ci sono **due ripetizioni**

Dopo aver capito come si calcola **m**, vediamo come funziona l'algoritmo

Esso funziona attraverso **5 step**:

1. **plurali e partecipi passati**: ogni volta che c'è **"sses"**, si sostituisce con **"ss"**, mentre il **gerundio** si gestisce **eliminando "ing"**
2. se la parola è **m > 0**, i **suffissi "ousness" e "ational"** diventano rispettivamente **"ous"** e **"ate"**
3. se la parola è **m > 0**, il **suffisso "icate"** diventa **"ic"**
4. se la parola è **m > 1** e **termina** con **"al"** o **"ance"**, questi vengono **rimossi**
5. se la parola è **m > 1** e **termina** con una **vocale**, allora questa viene **eliminata**, mentre se **termina** con una **doppia consonante**, ne viene **rimossa una**

In Python, per utilizzare quest'algoritmo, andiamo ad importare la classe `PorterStemmer` da nltk e il metodo che effettua lo **stemming** è `stemmer` che si invoca su un oggetto della classe

Snowball Stemming

È uno stemmer alternativo, più potente di quello di Porter

LEMMATIZATION

Il database morfologico più popolare utilizzato per fare lemmatization è **WordNet**

Grazie a questi database possiamo sapere per ogni parola se si tratta di un aggettivo, un verbo, ecc e sapere la **radice**

In generale, le parole sono composte da:

- **stem**: sono il core
- **affixes**: parti dello stem che hanno funzioni grammaticali

In nltk è presente la classe `WordNetLemmatizer` e per applicare la lemmatization si utilizza il metodo `lemmatize`

Esempio:

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
words=['John', 'and', 'Mike', 'are', 'going', 'to', 'school', 'with', 'the
ir', 'friends']
lemma=[lemmatizer.lemmatize(word) for word in words]
print(lemma)

>>> ['John', 'and', 'Mike', 'are', 'going', 'to', 'school', 'wit
h', 'their', 'friend']
```

Ciò che notiamo è che l'unica parola che ha subito una modifica è stata **"friends"** ed è diventata **"friend"**

In generale, un **lemmatizer** da solo **NON funziona**, perché ha bisogno di sapere la **Part-of-Speech (POS)** associata alla parola

Per cui la funzione `lemmatize` accetta un **secondo parametro**, il quale serve a dire all'algoritmo di lemmatization se una **parola** è un **nome, aggettivo, ecc.**

```
lemmatizer.lemmatize('going',pos='v') # returns 'go'  
lemmatizer.lemmatize('went',pos='v') # returns 'go'  
lemmatizer.lemmatize('computing',pos='n') # returns 'computing'  
lemmatizer.lemmatize('computing',pos='v') # returns 'compute'
```

Per trasformare per bene un testo dobbiamo seguire due passi:

- identificare le Part-of-Speech della frase
- eseguire la lemmatizzazione

Per identificare le POS, in nltk c'è `pos_tag` che permette di assegnare un'**etichetta** ad ogni **parola**

Queste etichette sono all'interno di una lista e sono:

- **cc** = coordinating conjunction
- **jj** = aggettivo
- **vb** = verbo
- **to** = infinite
- **vbn** = participio passato di un verbo

In questo modo teniamo conto anche delle coniugazioni

In realtà, a noi serve capire se è verbo, aggettivo, nome, avverbio, quindi ci basta solo la **prima lettera** di queste **sigle**

Possiamo creare una funzione che **ritorna la prima lettera del tagging**, in modo da lemmatizzare correttamente

In generale, possiamo dire che la **lemmatization funziona meglio** rispetto allo stemming
Il problema della lemmatization è che i **database lessicali NON sono disponibili per ogni lingua** e, inoltre, è lenta se applicata su documenti grandi

Il deep learning ha trovato un modo per ovviare questo problema, cioè utilizzando **word embedding**, cioè attribuendo alle parole un significato semantico

Le parole simili saranno dei vettori molto vicini, simili

MODELLO DI BASE DELL'IR

Vedremo i modelli su cui erano basati i primi motori di ricerca

Quando parliamo di IR facciamo riferimento a:

- **documento**
- **query**

Un motore di ricerca ritorna una serie di documenti sulla base della query

Piccolo OT non tanto OT: Bing Copilot combina LLM con un motore di IR, cioè recupera informazioni della rete e li passa al modello di LLM

Torniamo a noi, dobbiamo capire come **rappresentare i documenti e la query** e poi come **matchare query e documenti**

Tipicamente i **motori di ricerca** rappresentano i **documenti** con una serie di **parole chiave**

L'approccio tradizionale prevede che le **keyword si assegnavano manualmente**
Invece, noi vogliamo **estrarle automaticamente** dal documento

COME FUNZIONA L'IR?

Si hanno dei **documenti** e un **information need (query)**

Ciò che è fondamentale avere è un meccanismo di **ranking** perché molto spesso si hanno tanti documenti poco rilevanti per una query

Un **modello** di IR è una **quadrupla** composta:

- **set logico di documenti**
- **set logico di query**
- **framework**
- **algoritmo di ranking**

Vocabolario: è l'**insieme** di **termini** che ha **cardinalità pari** al **numero totale** di **termini distinti** all'interno della collezione di documenti

In generale, **rimuovere stop word** e fare **stemming** o **lemmatization** riduce il vocabolario

Se ho un vocabolario di t parole, posso rappresentare **ciascun documento** come un **vettore di booleani** dove se l' i -esimo termine è pari a 1, allora il termine si trova nel documento

In pratica otteniamo una matrice, detta **MATRICE TERMINI-DOCUMENTI**

Questa ha il **numero di righe** pari al **numero di documenti** e **numero di colonne** pari al numero di **termini**

Una **proprietà** della matrice è che è una matrice **fortemente sparsa**

COME OTTENIAMO QUESTA MATRICE?

Si seguono i passi che abbiamo visto

Il primo è fare un po' di pulizia del testo, quindi rimuoviamo stop word, facciamo stemming o lemmatization, ecc

Oppure si può utilizzare un **vocabolario controllato**

MODELLO BOOLEANO

È un modello semplice che si basa sulla teoria degli insiemi e algebra booleana

Infatti, le **query** sono **espressioni logiche**

Si possono fare combinazioni di **and** e **or**, combinazioni dei termini

Il **vantaggio** del modello è che si possono scrivere delle espressioni più o meno complesse

Inoltre per ogni documento, si ottiene **true** o **false** come **risultato** rispetto all'espressione booleana

Il problema è che **manca il ranking**, quindi non so quale documento è migliore rispetto alla query

Con questo meccanismo, **NON** si ha nemmeno un **matching parziale**

Dovremmo definire noi un concetto di partial matching, cioè mettere tutti in or

Quindi ciò che si può verificare è che si potrebbero **ottenere troppi documenti** se metto **tutti in or** oppure **troppo pochi** mettendo i **termini in and**

Per risolvere questi problemi, ci sono modelli più avanzati

▼ Lezione 9 - 04/04/2024

Ripartiamo dalla **matrice termini-documenti**

Abbiamo visto che il primo modo per crearla è con una **matrice booleana**, cioè se il **termine appare** nel documento ci sarà **1**, **altrimenti 0**

Ciò ci permette di fare query abbastanza semplici, ma questo modello **NON** ci permette di fare un ranking dei risultati

TERM WEIGHTING

Ora andiamo a creare modelli diversi **attribuendo un'importanza ai termini** andandoli a **pesare**

Ciò è importante perché i termini di un documento non hanno uguale importanza per descrivere il contenuto

L'idea di questo meccanismo è che possiamo associare un valore, un **peso** $w_{i,j}$ ai **termini**

Quindi andremo a rappresentare **ogni documento** come un **vettore di pesi**

COME ASSEGNA MO I PESI?

Il modo più semplice è **attribuire** ad ogni termine un **peso** che indica la **frequenza con cui** quel termine **compare nel documento**

Indichiamo con F_i la frequenza di un termine lungo tutti i documenti

Possiamo pensare che **se un termine compare tante volte**, allora **NON** ci permette di **identificare un documento**, **NON** aiutano a **discriminare**

Invece, i **termini** che **hanno una bassa frequenza** sono **utili?** Dipende

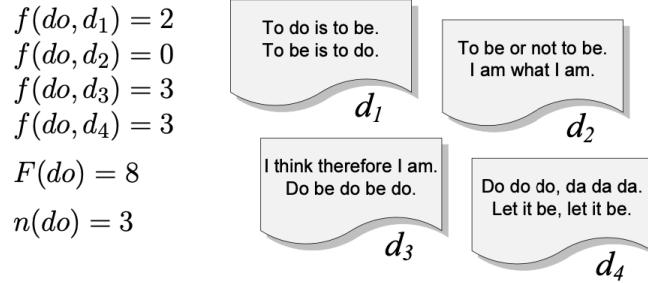
Esempio: se devo fare una ricerca per trovare il nome di una scheda grafica e ho solo 2 documenti, è giusto che quel termine viene considerato
Se devo fare un classificatore invece è poco utile

Però in generale, un **termine** che **compare molte volte** rischia di **NON** essere tanto **utile**

Quindi considerare la **frequenza** di un termine rischia di essere una cosa **NON** particolarmente **utile**

Però, ciò che possiamo fare è **vedere in quanti documenti compare il termine**

Esempio:



"do" compare 8 volte e in 3 documenti

Se ho 4 documenti e un termine compare in 3 doc, si rischia che la ricerca utilizzando questo termine **NON** è tanto utile

Quindi la **frequenza secca NON funziona**

TF-IDF WEIGHTS

Utilizziamo un ulteriore indice che **penalizza il peso** dei termini che **compaiono in troppi documenti**

Questo è il meccanismo di pesatura che andremo tipicamente ad utilizzare

TF (term frequency)

Andiamo a modificare leggermente la definizione di **TF**. Ciò che facciamo è **abbattere** con una **trasformazione logaritmica** la frequenza, quindi diventa:

$$f_{i,j} = 1 + \log f_{i,j} \text{ se } f_{i,j} > 0$$

In questo modo, otteniamo dei numeri più bassi rispetto alle frequenze secche

Document exhaustivity: numero di termini indici assegnati ad un documento

Quindi la **TF** indica l'**importanza** di un termine in un documento, mentre **IDF** (inverse document frequency) indica la **specificità**

Un termine è tanto più specifico quanto più è piccolo il numero di documenti in cui compare

Trattare la specificità in maniera semantica significa che in base alla mia conoscenza della lingua decido quali sono i termini di interesse.

Il processo di stop word removal lavora in maniera semantica

IDF (Inverse Document Frequency)

È definito come:

$$\log \frac{N}{n_i}$$

dove

- n_i è in quanti documenti compare il termine i
- N è il **numero totale di documenti** della mia collezione

Questa formula fa capire quanto è specifico il termine, **è inversamente proporzionale al numero di documenti in cui compare**.

Se un **termine** ha **IDF** pari a 0 allora lo scartiamo.

TF-IDF

È la moltiplicazione dei due componenti, cioè un peso che va ad accoppare la **term frequency** e la **inverse document frequency**

$$w_{i,j} = (1 + \log f_{i,j}) * \log \frac{N}{n_i}$$

DOCUMENT LENGTH NORMALIZATION

Eventualmente potremmo **normalizzare** il rank del documento rispetto alla sua **lunghezza**, questo perché potremmo avere in collezione documenti troppo corti e altri troppo lunghi.

Questo è un problema perché documenti lunghi hanno più probabilità di essere recuperati da un sistema IR.

Un documento per noi altro non è che un vettore in uno spazio (in particolare spazio dei termini)

Il documento sarà un vettore orientato nello spazio dei termini e avrà un certo **modulo** per ogni direzione, pari al peso del termine.

Dato un vettore, come posso calcolarne la **lunghezza**?

La lunghezza del documento è data dalla norma del suo vettore:

$$\|\vec{d}\| = \sqrt{\sum_i^t w_{ij}^2}$$

THE VECTOR MODEL

Fino ad adesso abbiamo definito:

- com'è fatto un vettore
- qual è la lunghezza del documento

Vediamo come definire la **similitudine** di un documento in uno spazio vettoriale.

Utilizzare il **prodotto scalare** può risultare problematico, poiché è fortemente influenzato dalla lunghezza dei documenti.

Un'alternativa è impiegare la **distanza euclidea** (basata sul teorema di Pitagora), ma questa viene scartata per lo stesso motivo: è ancora più sensibile alla differenza di lunghezza tra i documenti rispetto al prodotto scalare.

COME SI FA?

Assegnamo ad un documento una **lista di pesi**

Il prof ha ridotto la rappresentazione del documento a un **vocabolario di due termini**.

Quindi, ogni documento avrà un peso per il **primo termine i** e un peso per il **secondo termine j**

Anche la **query sarà rappresentata in modo analogo**, con due pesi corrispondenti ai due termini del vocabolario

Per calcolare la **distanza** tra il documento e la query, si potrebbe utilizzare la **distanza euclidea** ma questo approccio presenta un problema: se il documento è molto lungo e la query è breve, la distanza risulterà grande, anche se i due vettori (documento e query) potrebbero essere simili nella direzione.

Se invece vogliamo un **indice di similitudine** che non sia influenzato dalla lunghezza del documento, possiamo usare il **coseno dell'angolo** tra i due vettori (**documento e query**).

In questo modo, otteniamo un indice di similitudine compreso tra **0 e 1**.

Il **coseno** di calcola facendo prodotto scalare normalizzandolo con il prodotto delle norme dei due vettori

$$\cos(\theta) = \frac{\vec{d}_j \cdot \vec{q}}{|\vec{d}_j| \times |\vec{q}|}$$

Il **prodotto scalare** tra il vettore del documento e quello della query **considera solo i termini che compaiono in entrambi**, ovvero quelli che hanno un peso in entrambi i vettori.

I termini che non appaiono in una delle due parti non contribuiscono al prodotto scalare.

I **pesi** assegnati ai termini sono basati sulla misura **tf-idf**.

Un'**assunzione importante** che facciamo in questo modello è che i termini siano **mutuamente ortogonali** tra loro.

Questo significa che consideriamo ogni termine come una **dimensione indipendente** nello spazio vettoriale: in altre parole, **tutti i termini sono indipendenti tra loro**

Assumiamo inoltre che la **query** non va a correggere l'idf, questo perché se ho tanti documenti e la query è unica essa non influenza in modo significativo il peso.

Vantaggi: permette un matching parziale, è facile da interpretare (similitudine del coseno), e include la normalizzazione rispetto alla lunghezza del documento.

Svantaggi: non considera l'ordine e il contesto delle parole, limitando la capacità di comprendere la semantica del testo

VEDIAMO IN PYTHON

Utilizziamo due librerie:

- `nltk`
- `scikit-learn`

La prima operazione consiste nell'istanziare un oggetto di tipo `CountVectorizer`.

Questa classe consente di rappresentare i documenti con le **frequenze secche (tf)** delle parole, ovvero quante volte un termine appare in un documento.

Supponiamo di avere una lista di documenti, dove ogni documento è una stringa

La funzione `fit_transform()` prende i documenti e **crea** un modello del `CountVectorizer`, cioè una **matrice in cui ogni termine è pesato con la frequenza** con cui compare nel documento

`shape` ci dà le dimensioni del modello, cioè la coppia numero di documenti - numero di termini

`get_features_names_out` restituisce il vocabolario

A questo punto, possiamo eliminare le **stopword**, ovvero le parole comuni che non sono rilevanti per il contenuto (es. "il", "e", "che")

Notiamo che il `CountVectorizer` esegue **automaticamente** la **tokenizzazione** (separazione delle parole), ma non esegue lo **stemming**.

Dato che manca lo stemming nel `CountVectorizer`, dobbiamo definire una nostra funzione di **tokenizzazione** personalizzata che includa questa operazione.

`pruned` scorre i token e aggiunge alla lista i token che iniziano con un carattere alfanumerico e non è presente nelle stopword, dopodiché calcola gli stem

Quando si inizializza il `CountVectorizer` si può passare il proprio tokenizzatore
Dopo queste operazioni il numero di termini è **ridotto**.

Ora vogliamo calcolare la **matrice di similitudine**, è un'operazione facile da fare grazie alla funzione `cosine` importata prima.

Otteniamo così una matrice quadrata e simmetrica.

Vediamo come usare TF-IDF

Per rappresentare i documenti usando **TF-IDF** utilizziamo la classe `TfidfVectorizer` di `scikit-learn`

Questo strumento consente di pesare i termini non solo in base alla loro frequenza nei documenti, ma anche tenendo conto della loro rarità rispetto all'intero corpus, penalizzando i termini troppo comuni.

Come prima, possiamo passare una funzione di **tokenizzazione** personalizzata a `TfidfVectorizer`

Dopo averlo configurato, utilizziamo la funzione `fit_transform()` per creare il modello TF-IDF basato sui documenti.

Vediamo ora come trattare una query esterna

La query è: "racing game".

La prima cosa da fare è rappresentare la query all'interno del modello e per farlo utilizziamo il metodo `transform()` del `TfidfVectorizer`.

Questo metodo trasforma la query nel **vettore TF-IDF corrispondente**, ma **non modifica** il modello già creato, a differenza di `fit_transform()`, che crea e addestra il modello allo stesso tempo.

Per calcolare la similitudine tra la query e i documenti nel modello, utilizziamo la funzione `cosine_similarity()`.

A differenza del calcolo tra documenti (che restituisce una **matrice simmetrica**), quando calcoliamo la similitudine con una query, otteniamo una **matrice non simmetrica**.

La prima riga rappresenta il peso delle parole della query nel vocabolario, mentre le righe successive rappresentano la similitudine tra la query e ciascun documento.

TORNIAMO ALLA TEORIA

PROBLEMI NELL'IMPLEMENTAZIONE A SPAZI VETTORIALI

Il primo problema da affrontare sono le rappresentazioni **sparse**, cioè abbiamo vettori **grandi** con **poche** informazioni.

Dovremmo pensare a rappresentazioni più efficienti.

Abbiamo varie possibilità:

- rappresentare i vettori **come liste concatenate**, ma per fare una ricerca avremo complessità $O(N^2)$
- rappresentazione dei token mediante un **balanced binary tree**, dove la complessità è data dalla profondità dell'albero.
- utilizzare un **HashMap**, così si ha un tempo **costante** per cercare un termine nella **struttura dati**, mentre complessità **lineare** per l'inserimento.

Così si risolve il problema delle **matrici sparse**, ma resta il problema di dover confrontare una query con tutti i documenti

CONFRONTO CON TUTTI I DOCUMENTI

La rappresentazione della matrice termini-documenti che si utilizza è detta **a indici invertiti**

La struttura dati è un **dictionary (HashMap)** dove:

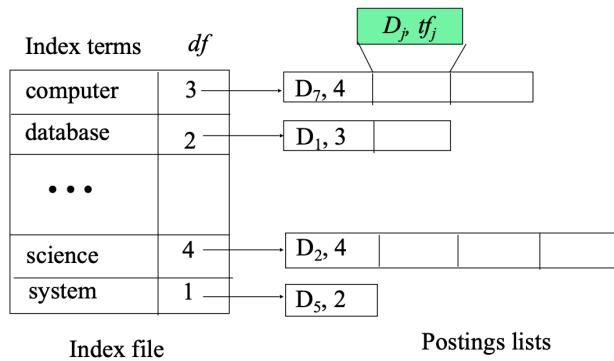
- le **chiavi** sono i singoli termini del vocabolario
- i **valori** sono la **df (document frequency)** e un **puntatore** ad una lista di tuple che conterranno i documenti in cui appare il termine e il peso.

La **df** è la lunghezza della lista.

Con questo metodo posso confrontare i termini solo nei documenti in cui compaiono.

Esempio: se ho 2 parole, devo scorrere solo 2 liste

Inverted Index



▼ Lezione 10 - 08/04/2024

Abbiamo visto che invece di rappresentare ogni documento con un vettore, ora rappresentiamo i dati con un **hashmap** che indicizza i termini e in corrispondenza di ognuno abbiamo una lista di tutti i documenti che contengono quella parola.

Ogni elemento della lista sarà una tupla che conterrà il documento in cui compare la parola e il peso nello stesso.

Tutti i token che **NON** sono **né nella query né nel dizionario**, non hanno effetto

Quindi il **prodotto scalare** viene costruito andando a recuperare solo le parole che ci sono nella query dalla matrice dei termini inversa

Il vantaggio di questo meccanismo è che tipicamente la query è molto corta, per cui bisogna recuperare informazioni per un numero limitato di parole

Quando si deve fare ciò, i **prodotti scalari** si costruiscono in **maniera incrementale**

In generale, il **tempo di retrieval sarà lineare** perché devo costruire il prodotto scalare di tutti gli elementi che ci sono nella lista associata al termine

Esempio: 1000 documenti e query di 2 parole \Rightarrow complessità = $O(2000)$

Con una **matrice convenzionale**, bisognava calcolare il **coseno tra la query per ogni documento**, quindi un vettore di lunghezza pari al vocabolario per ogni documento

La complessità sarebbe stata $O(V * N)$ dove V = dimensione del vocabolario e N = numero di documenti

Quando si processa la query, il coseno viene calcolato in modo incrementale processando le parole una alla volta.

Vediamo implementazione matrice inversa in Python

La prima implementazione è con la classe `HashingVectorizer` in `scikit-learn`

Funziona come `CountVectorizer` ma non usa tf-idf perché renderebbe la rappresentazione statica.

Un'altra implementazione è con la libreria `gensim`

`gensim` potrebbe dare problemi su alcune architetture

Per risolvere: disinstallare `scipy` e `gensim`, successivamente installare prima `scipy 1.10.1` e poi `gensim`

```
pip install scipy==1.10.1
```

In generale, per eseguire una query bisogna:

- tokenizzare la query
- trasformarla in una bag of word

N-grammi

Gli n-grammi rappresentano il primo passo verso modelli più sofisticati, come quelli utilizzati nelle reti neurali.

Fino ad ora, abbiamo rappresentato i documenti come un insieme di parole **indipendenti** tra loro.

Ora, ogni parola assume un significato che viene catturato osservando anche le parole circostanti.

Usare gli **n-grammi** significa che il mio dizionario sarà composto da:

- tutte le parole
- tutte le possibili sequenze di parole di una certa lunghezza

Due parole formano un bi-gramma se sono adiacenti.

Come implementiamo questo meccanismo? È relativamente semplice.

Sia con `CountVectorizer`, sia con `TfidfVectorizer` che con `HashingVectorizer` si può specificare, durante l'inizializzazione del modello, il numero minimo e massimo di n-grammi da formare tramite il parametro `ngram_range`. Di default, il parametro è impostato a (1,1).

QUERY REFORMULATION

La riformulazione della query è un meccanismo molto utilizzato nel **reinforcement learning**.

Questo processo riflette un comportamento naturale che tutti noi, inconsciamente, adottiamo: spesso modifichiamo la query per ottenere una risposta migliore.

Come funziona?

Eseguo una query e ottengo un certo numero di documenti. Dopo aver ricevuto una lista ordinata (**ranked list**) dei documenti rispetto alla query, per ognuno di questi assegno un valore booleano, indicando se il documento è rilevante o meno.

A questo punto, applico la **formula di Rocchio**, un semplice meccanismo di **ripesatura**:

$$Q_1 = \alpha \cdot Q_0 + \frac{\beta}{n_1} \sum_{i=1}^{n_1} R_i - \frac{\gamma}{n_2} \sum_{i=1}^{n_2} S_i$$

dove:

- Q_0 è il vettore della query iniziale
- R_i è il vettore del documento rilevante i-esimo
- S_i è il vettore del documento non rilevante i-esimo
- n_1 è il numero di documenti rilevanti scelti

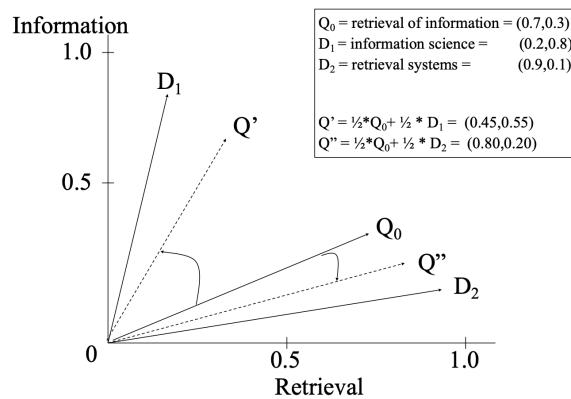
- n_2 è il numero di documenti non rilevanti scelti
- α, β e γ sono i pesi dati ai termini (spesso $\beta = 0.75$ e $\gamma = 0.25$)

I **documenti** sono rappresentati come **vettori**, così come la **query**.

La **query** viene **sommata** in modo **positivo** con i **documenti rilevanti** e in modo negativo con quelli non rilevanti.

Si tratta quindi di **somme vettoriali**: penalizzo i termini contenuti nei documenti non rilevanti.

Da un punto di vista **visivo**, cosa succede?



Supponiamo di avere la query Q_0 e due documenti D_1 e D_2 , se considero D_1 rilevante e D_2 no, aggiungo a Q_0 il vettore di D_1 .

In questo modo, la query si sposta verso D_1 , ottenendo una query Q' che è una via di mezzo tra la query originale e i documenti rilevanti rispetto alla query iniziale.

Questo sistema funziona bene quando si implementa un sistema interattivo, in cui, dopo aver eseguito una query, l'utente può indicare tramite pulsanti se i documenti sono rilevanti o meno, e poi rieseguire la query.

LATENT SEMANTIC INDEXING

Problemi: sinonimia e polisemia.

- **Sinonimia:** usiamo termini diversi per esprimere gli stessi concetti
Il problema è che potremmo fallire nel recupero di documenti potenzialmente rilevanti. Una prima soluzione potrebbe essere quella di usare un **tesauro**, aggiungendo sinonimi. Tuttavia, questo potrebbe portare al recupero di più documenti, inclusi alcuni non rilevanti
- **Polisemia:** una stessa parola può assumere più significati (es. *book*)

Vorremmo che i documenti rappresentassero concetti e non solo un insieme di termini.

La prima tecnica che vediamo è il **Latent Semantic Indexing**

Questa tecnica si basa sul **rappresentare la conoscenza** non più come una **relazione** tra documenti e termini, ma **tra documenti e concetti**

Lavoriamo quindi in uno spazio diverso e anche più piccolo.

Questo spazio ridotto permette di catturare relazioni che non sarebbero evidenti nello spazio vettoriale convenzionale, dove ogni termine è trattato come un evento indipendente.

SVD: Decomposizione al valore singolare

L'SVD è una trasformazione che consente di spostare i dati dallo spazio dei termini allo spazio dei concetti. Prende in input una matrice e la decomponete in tre matrici:

- **X:** matrice **termini-documenti**
- **U:** di dimensione $m * k$, chiamata **matrice dei documenti**
- **Sigma:** matrice diagonale $k * k$ che contiene i valori singolari (**autovalori**) che rappresentano il rango della matrice X. Se nella matrice originale ci sono vettori linearmente dipendenti, il rango diminuirà.
- **V:** di dimensione $k * n$, dove k è il rango e n è il numero di termini nel vocabolario.
Questa matrice mette in relazione i termini con i concetti.

Le matrici **U** e **V** sono costituite da **autovettori**, quindi contengono solo vettori linearmente indipendenti.

In questo modo, i k **concetti dello spazio risultano linearmente indipendenti**, mentre i termini sono legati tra loro

Rappresentiamo quindi i documenti tramite fattori linearmente indipendenti

In pratica, si parte dalla **matrice termini-documenti X** e questa viene decomposta in **3 matrici**

$$X = U\Sigma V^T$$

- Σ is a $k \times k$ diagonal matrix containing singular values
 - where k is the rank of X
- U ($m \times k$) and V ($k \times n$) contains eigenvectors, i.e., linearly independent vectors

Uno dei problemi iniziali sarà che il numero di concetti potrebbe essere grande, come il rango della matrice.

Successivamente, ci accorgeremo che alcuni concetti avranno un peso basso.

Dovremo quindi capire quali sono i concetti importanti e scartare quelli meno rilevanti.

Esempio:

LSI Example

Consider a collection of documents:

- d1: Indian government goes for open-source software
- d2: Debian 3.0 Woody released
- d3: Wine 2.0 released with fixes for Gentoo 1.4 and Debian 3.0
- d4: gnuPOD released: iPod on Linux... with GPLed software
- d5: Gentoo servers running at open-source mySQL database
- d6: Dolly the sheep not totally identical clone
- d7: DNA news: introduced low-cost human genome DNA chip
- d8: Malaria-parasite genome database on the Web
- d9: UK sets up genome bank to protect rare sheep breeds
- d10: Dolly's DNA damaged

I primi 5 documenti parlano di sistemi software mentre gli altri di DNA

Prendiamo **matrice dei documenti** e notiamo che nel documento 9 non c'è la parola DNA anche se se ne parla

Si ottengono similitudini pari a 0 che però non sono rappresentative della realtà

LSI Example: term-documents matrix

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
open-source	1	0	0	0	1	0	0	0	0	0
software	1	0	0	1	0	0	0	0	0	0
Linux	0	0	0	1	0	0	0	0	0	0
released	0	1	1	1	0	0	0	0	0	0
Debian	0	1	1	0	0	0	0	0	0	0
Gentoo	0	0	1	0	1	0	0	0	0	0
database	0	0	0	0	1	0	0	1	0	0
Dolly	0	0	0	0	0	1	0	0	0	1
sheep	0	0	0	0	0	1	0	0	0	0
genome	0	0	0	0	0	0	1	1	1	0
DNA	0	0	0	0	0	0	2	0	0	1

Per capire **l'importanza di una parola** rispetto al concetto dobbiamo considerare il valore assoluto dalla matrice U, invece dalla matrice V vediamo l'importanza di un documento rispetto ai concetti.

Andando a ricalcolare la similarità si nota che scegliendo **k basso**, quindi pochi concetti, si tende a fare un forte clustering di documenti.

Ci sono delle **metriche** che permettono di calcolare la coerenza di questo spazio e quindi stabilire qual è il valore ottimale da prendere

Pro di LSI:

- gestisce sinonimia e onomimia (si può anche eliminare lo stemming, è meno importante)
- otteniamo una maggiore similitudine tra documenti della stesso cluster
- otteniamo una minore similitudine tra doc di cluster diversi

Contro:

- più costoso
- difficoltà nell'aggiunta di un nuovo documento (dovrei usare trasformazioni matematiche complesse per fare il folding di un documento in uno spazio già creato)

In Python si lavora con `gensim`

La differenza rispetto agli script di prima è che la funzione principale è `LsiModel`, a cui bisogna passare il corpus, il numero di topic e il dizionario.

Per eseguire la query, la devo traslare nello spazio dei concetti, quindi fare il folding

▼ Lezione 11 - 11/04/2024

LSI IN PYTHON

Abbiamo utilizzato `gensim` per implementare LSI

Il cuore del processo è la creazione del modello `LsiModel`, in cui passiamo il corpus trasformato in TF-IDF e il numero di topic da considerare.

Questo modello ci permette di misurare la **similitudine tra documenti** anche se non contengono parole in comune.

Tuttavia, un problema di LSI è che i documenti vengono proiettati in uno **spazio decomposto**, quindi la **query deve essere anch'essa traslata in quello spazio**, un processo chiamato **folding**

Prima rappresentiamo la query come **bag of words** e poi la convertiamo nello spazio LSI.

Per ogni documento otteniamo un certo numero di tuple (una per ogni topic), dove ogni tupla contiene l'ID del topic e lo score associato a quel topic.

I **valori** nella **matrice documenti-concetti** vanno considerati in **valore assoluto**.

Un modo per verificare se stiamo considerando il numero giusto di topic è cercare di capire il significato di ogni topic, identificando le parole che lo rappresentano.

Questo processo richiede **sperimentazione**, quindi è possibile fare un'analisi:

- **quantitativa** utilizzando una metrica chiamata **coerenza**.
- **qualitativa** esaminando le parole che compongono i topic.

Se esageriamo con il numero di topic, noteremo la presenza di topic con **set di parole duplicati**.

L'indice che ci fornisce un'idea quantitativa è la **coerenza**. Idealmente, dovremmo scegliere il valore massimo di coerenza, ma ciò potrebbe portare a una frammentazione eccessiva dei topic.

Per questo motivo, è spesso preferibile fermarsi al **punto di flesso** nel grafico della funzione di coerenza.

La coerenza si calcola iterando su un ciclo che varia il numero di topic.

Ad ogni iterazione, creiamo un modello LSI sullo stesso corpus, ma con un numero diverso di topic.

Poi, creiamo la funzione `CoherenceModel`, che prende in input il modello, il testo dei documenti, il dizionario, e offre l'opzione di scegliere il metodo di calcolo della coerenza.

Infine, otteniamo i valori di coerenza, li memorizziamo in un array e li visualizziamo tramite un grafico.

Per ottenere una buona coerenza, nell'**esempio** sulle slide, il numero ottimale di topic potrebbe essere **3**.

Tuttavia, con il calcolo della coerenza effettuato tramite il metodo **u_mass** (come parametro del modello di coerenza), il grafico potrebbe risultare più rappresentativo.

Oltre a LSI, esiste un altro modello per analizzare documenti basati sui **topic**: si tratta di **LDA (Latent Dirichlet Allocation)**

La differenza sostanziale è che mentre **LSI** effettua una trasformazione mediante **decomposizione a valori singolari**, **LDA** adotta un **approccio probabilistico**

Con LDA, **ogni parola** nel dizionario **ha una certa probabilità di apparire in un topic**, quindi ogni topic sarà caratterizzato da parole più importanti rispetto ad altre, cioè quelle che avranno una probabilità più alta di comparire.

I documenti vengono rappresentati come distribuzioni di probabilità di topic, il che significa che ogni topic in un documento avrà una probabilità associata.

LDA è implementato, tra le altre librerie, in `gensim`.

CLASSIFICATORI DI MACHINE LEARNING

Vedremo alcune tecniche di **machine learning** utilizzate per la classificazione del testo.

Una delle tecniche più semplici è l'uso di un **classificatore bayesiano**, che sfrutta il **teorema di Bayes**.

Esempi di classificazione del testo includono lo **spam detector** e la **sentiment analysis**.

Molti meccanismi di machine learning e statistici utilizzano le probabilità, poiché molti fenomeni non permettono una classificazione netta, ma piuttosto una classificazione probabilistica.

NAIVE BAYES

Come vengono calcolate queste probabilità?

Tipicamente, le probabilità si calcolano sulla base di eventi passati. I modelli stimano quindi una probabilità basandosi sui dati storici.

Un **classificatore bayesiano** utilizza i dati di addestramento per calcolare le probabilità.

Le applicazioni sono moltissime: classificazione del testo, **intrusion detection** (sicurezza delle reti), applicazioni mediche, e molte altre

Dove viene applicato?

Questi modelli sono particolarmente efficaci nel contesto del testo, dove ci sono molte **features** (cioè, le parole). La teoria bayesiana non stima una vera e propria probabilità, ma una **verosimiglianza**, che è simile alla probabilità, poiché viene calcolata come rapporto tra eventi favorevoli e eventi totali, ma sui dati noti.

Terminologia:

- **Verosimiglianza**: rapporto tra eventi osservati e il numero di prove (trial).
- **Trial**: non rappresenta i casi possibili, ma i tentativi o prove effettivamente eseguite.

Vale la legge del complemento:

$$P(A) = p$$
$$P(A') = 1 - p$$

Vediamo come si può utilizzare la **probabilità condizionata** per fare classificazione.

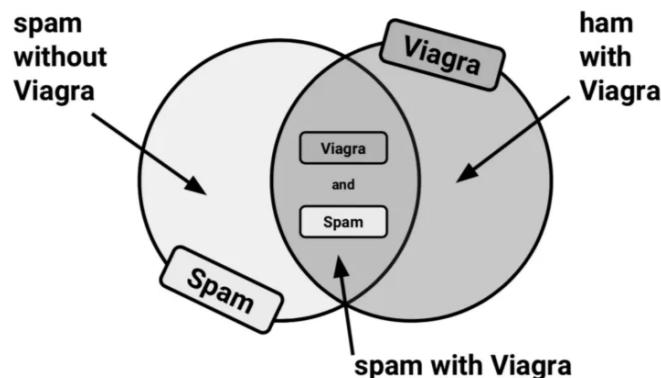
Supponiamo che, sulla base dei messaggi passati, l' **80%** dei messaggi sia classificato come **ham** e il **20%** come **spam**. Ora vorremmo trovare un modo per classificare automaticamente i messaggi di spam.

Questo può essere fatto calcolando una **probabilità composta**.

Ad **esempio**, il professore ha osservato che il **5%** delle email conteneva la parola "viagra", una parola che può essere indicativa di spam.

Tuttavia, non è detto che tutte queste email siano spam. Ora dobbiamo capire quante email di spam e ham contengono la parola "viagra".

Possiamo fare un'**analisi insiemistica** per capire la relazione tra la presenza della parola "viagra" e la probabilità che l'email sia spam o ham



Se l'intersezione è grande, significa che la parola "viagra" è un buon indicatore di spam. Dobbiamo fare attenzione ai **falsi positivi** e **falsi negativi**.

Supponiamo che il **20%** dei messaggi sia spam e che il **5%** contenga "viagra".

Vogliamo calcolare l'**overlap**

Potremmo calcolare una **probabilità composta**, cioè la probabilità che un messaggio sia spam e contenga "viagra".

La cosa più semplice sarebbe calcolare il prodotto delle probabilità, ma ciò è valido solo se i due eventi sono **indipendenti**.

Due eventi sono considerati indipendenti se il verificarsi di uno non influisce sull'altro.

Tuttavia, ci sono casi in cui gli eventi sono **dipendenti**; in tal caso, possiamo utilizzare il **teorema di Bayes**:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

La bellezza del teorema di Bayes è che possiamo riscrivere la formula come segue:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$$

Facciamo questa operazione perché non conosciamo $P(A|B)$ (la probabilità che un documento sia spam dato che contiene "viagra"), ma conosciamo $P(B | A)$ (la probabilità che un documento contenga "viagra" ed è spam)

Diamo i nomi ai 4 elementi del teorema di bayes:

- $P(A) = P(\text{Spam})$ è l'evidenza che i messaggi precedenti erano spam: **prior probability**
- $P(B|A) = P(\text{Viagra}|\text{Spam})$ è l'evidenza che viagra sia in spam: **likelihood (verosimiglianza)**
- $P(B) = P(\text{Viagra})$ probabilità che la viagra sia in un messaggio di spam

Quindi abbiamo:

$$P(Spam|Viagra) = \frac{P(Viagra|Spam)P(spam)}{P(Viagra)}$$

Costruiamo una tabella delle frequenze:

		Viagra		
Frequency		Yes	No	Total
spam		4	16	20
ham		1	79	80
Total		5	95	100

Ora possiamo calcolare il likelihood

Dividiamo i numeri rispetto al totale dei documenti

Likelihood table

		Viagra					Viagra		
Frequency		Yes	No	Total	Likelihood		Yes	No	Total
spam		4	16	20	spam		4 / 20	16 / 20	20
ham		1	79	80	ham		1 / 80	79 / 80	80
Total		5	95	100	Total		5 / 100	95 / 100	100

A questo punto possiamo fare i calcoli:

$$P(Viagra = Yes|spam) = 4/20 = 0.2$$

$$P(spam \cap viagra) = P(Viagra|spam) * P(spam) = (\frac{4}{20}) * (\frac{20}{100}) = 0.04$$

La probabilità calcolata come eventi indipendenti era 0.01, quindi stavamo sottostimando la probabilità

Andiamo ora a calcolare quello che ci serve davvero, cioè la probabilità a posteriori:

$$P(\text{spam}|\text{viagra}) = (4/20) * (20/100)/(5/100) = 0.80$$

Quindi questo classificatore mi dice che la probabilità di un messaggio che contiene viagra è di spam è dell'80%

ALGORITMO DEL NAIVE BAYES CLASSIFIER

Perché si chiama "naive"?

Perché **assume** che tutte le **parole** presenti in un documento **siano equamente importanti e indipendenti tra loro**, anche se, come abbiamo visto nei modelli di **Information Retrieval**, questo non è sempre vero.

Il modello **bayesiano** funziona quindi come un **modello a spazi vettoriali classico**, poiché considera le parole come indipendenti tra loro

Vantaggi:

- Funziona bene anche con documenti rumorosi e incompleti.
- Rispetto ad algoritmi più complicati, è efficace anche con piccoli set di addestramento.
- Il calcolo della probabilità stimata è relativamente semplice.

Svantaggi:

- Alcune delle assunzioni che facciamo non si verificano sempre nella realtà.
- Questo modello funziona bene quando le caratteristiche sono del tipo **sì/no** (cioè presenti o assenti), mentre non è altrettanto efficace quando si tratta di caratteristiche di tipo numerico.

Prima abbiamo visto come funziona l'algoritmo con una sola parola. Ora vediamo come funziona con un dizionario.

Prendiamo un **esempio** semplice con altre tre parole e facciamo lo stesso esercizio di prima su tutte le parole, ottenendo la seguente tabella:

Likelihood	Viagra (W_1)		Money (W_2)		Groceries (W_3)		Unsubscribe (W_4)		Total
	Yes	No	Yes	No	Yes	No	Yes	No	
spam	4 / 20	16 / 20	10 / 20	10 / 20	0 / 20	20 / 20	12 / 20	8 / 20	20
ham	1 / 80	79 / 80	14 / 80	66 / 80	8 / 80	71 / 80	23 / 80	57 / 80	80
Total	5 / 100	95 / 100	24 / 100	76 / 100	8 / 100	91 / 100	35 / 100	65 / 100	100

Per capire se un documento è spam o meno, bisogna calcolare per questo documento la **posterior probability**.

Supponiamo che il documento contenga le parole "viagra" e "unsubscribe", ma non "money" e "groceries".

Dobbiamo calcolare la probabilità condizionata di spam dato l'intersezione delle probabilità delle parole.

Tuttavia, questo modello ha un altro piccolo problema: se una parola presente nel documento non è presente nei dati passati, la formula va in crisi.

Usiamo la **naive assumption** cioè assumiamo che gli eventi siano indipendenti tra loro e calcoliamo il numeratore della formula come una semplice moltiplicazione.

Il denominatore è costante e quindi può essere ignorato.

- The conditional probability of spam is:

$$P(\text{spam} | W_1 \cap W_2 \cap W_3 \cap W_4) \propto P(W_1 | \text{spam}) P(W_2 | \text{spam}) P(W_3 | \text{spam}) P(W_4 | \text{spam}) P(\text{spam})$$

La stessa cosa si può fare per la probabilità di ham

Likelihood	Viagra (W_1)		Money (W_2)		Groceries (W_3)		Unsubscribe (W_4)		Total
	Yes	No	Yes	No	Yes	No	Yes	No	
spam	4 / 20	16 / 20	10 / 20	10 / 20	0 / 20	20 / 20	12 / 20	8 / 20	20
ham	1 / 80	79 / 80	14 / 80	66 / 80	8 / 80	71 / 80	23 / 80	57 / 80	80
Total	5 / 100	95 / 100	24 / 100	76 / 100	8 / 100	91 / 100	35 / 100	65 / 100	100

Viagra=Yes, Money=No, Groceries=No, Unsubscribe=Yes

- Overall likelihood of spam:
 $(4/20)*(10/20)*(20/20)*(12/20)*(20/100)=0.012$
- Overall likelihood of ham:
 $(1/80)*(66/80)*(71/80)*(23/80)*(80/100)=0.002$
- $0.012/0.002=6$, hence this message is six times more likely to be spam than ham

Tuttavia, queste **likelihood** non sono significative da sole.

Pertanto, calcolo il **rapporto** tra la **likelihood di spam e quella di ham**, ottenendo che è **6 volte più probabile trovare spam** rispetto ad ham.

Per convertire questo rapporto in probabilità, dobbiamo considerare nuovamente il denominatore.

Il classificatore avrà una soglia che permetterà di determinare se il documento è spam o ham.

Il problema è che la probabilità è stimata come un prodotto di probabilità, quindi sarebbe **problematico** se uno dei termini andasse a **0**.

STIMATORE DI LAPLACE

Vediamo un altro **esempio**: un documento che contiene tutte e quattro le parole.

Dobbiamo calcolare la **likelihood** di spam.

Let's compute p(spam) and p(ham)

Likelihood	Viagra (W ₁)		Money (W ₂)		Groceries (W ₃)		Unsubscribe (W ₄)		Total
	Yes	No	Yes	No	Yes	No	Yes	No	
spam	4 / 20	16 / 20	10 / 20	10 / 20	0 / 20	20 / 20	12 / 20	8 / 20	20
ham	1 / 80	79 / 80	14 / 80	66 / 80	8 / 80	71 / 80	23 / 80	57 / 80	80
Total	5 / 100	95 / 100	24 / 100	76 / 100	8 / 100	91 / 100	35 / 100	65 / 100	100

- Using the likelihood table, we compute the likelihood of spam:
 $(4/20)*(10/20)*(0/20)*(12/20)*(20/100)=0$
- Similarly, the likelihood of ham:
 $(1/80)*(14/80)*(8/80)*(23/80)*(80/100)=0.00005$
- The probability of spam is:
 $0/(0+0.00005)=0$
- The probability of ham is:
 $0.00005/(0+0.00005)=1$

Se calcoliamo il likelihood di "groceries", otteniamo **0** (DISASTRO).

Di conseguenza, quando convertiamo queste **likelihood** in probabilità di spam, otteniamo che la probabilità di spam è **0**, e quindi la probabilità di ham sarebbe **1**.

Questo rappresenta un problema del modello, poiché non ha alcun senso.

Per affrontare questo problema, utilizziamo lo **stimator di Laplace**.

Invece di dire che una parola compare **0** volte, sostituiamo questo **0** con un valore molto piccolo. In questo modo, diamo un peso quasi nullo, ma che ci permette di evitare che la probabilità vada a **0**.

Vediamo come si modifica il modello: aggiungiamo **1** a tutti i numeratori per calcolare il **likelihood**.

Questa operazione deve essere effettuata su tutti i termini, altrimenti daremmo troppa importanza a un termine specifico.

Il denominatore non sarà più **20**, ma **24** (poiché abbiamo **4** termini nell'esempio).

La **posterior probability** non cambia.

Ora, il likelihood di spam è **4 volte** più grande di ham. Convertendo in probabilità, otteniamo che la probabilità di spam è **0.80** e quella di ham è **0.20**.

Let's recompute the number

		Viagra (W_1)		Money (W_2)		Groceries (W_3)		Unsubscribe (W_4)		Total
Likelihood	Yes	No	Yes	No	Yes	No	Yes	No		
spam	4 / 20	16 / 20	10 / 20	10 / 20	0 / 20	20 / 20	12 / 20	8 / 20	20	
ham	1 / 80	79 / 80	14 / 80	66 / 80	8 / 80	71 / 80	23 / 80	57 / 80	80	
Total	5 / 100	95 / 100	24 / 100	76 / 100	8 / 100	91 / 100	35 / 100	65 / 100	100	

- Likelihood of spam:
 $(5/24)*(11/24)*(1/24)*(13/24)*(20/100)=0.0004$
- Similarly, the likelihood of ham:
 $(2/84)*(15/84)*(9/85)*(24/84)*(80/100)=0.0001$
- The probability of spam is:
 $0.0004/(0.0004+0.0001)=0.80$
- The probability of ham is:
 $0.0001/(0.0004+0.0001)=0.20$

COME USARE FEATURE NUMERICHE

In teoria, per costruire tabelle di likelihood è necessario avere features categoriche.

Per questo si ricorre a un espediente: una libreria discretizza i nostri dati.

Supponiamo, ad esempio, di avere come feature aggiuntiva l'orario del giorno in cui riceviamo un'email, oltre alle parole contenute nel testo.

Posso aggiungere delle feature booleane che esprimono l'orario del giorno. In teoria, dovrei aggiungere tante feature booleane quanti sono i valori rappresentabili, oppure potrei decidere un livello di discretizzazione.

Ad esempio, possiamo dividere l'orario del giorno in quattro variabili che indicano notte, mattina, pomeriggio e sera.

Esempio: classificazione di SMS come spam o ham

La funzione `shape` ci dà il numero di righe e colonne del dataset

La funzione di preprocessing effettua le seguenti operazioni:

- conversione del testo in minuscolo (lowercase)
- rimozione di spazi multipli
- rimozione delle stopword
- eliminazione dei caratteri di punteggiatura

- eliminazione delle parole che sono numeri
- la funzione `strip_short` elimina tutte le parole con lunghezza inferiore a una certa soglia
- applicazione dello stemming

Rimuovere parole corte può essere utile nella classificazione, ma non tanto nell'information retrieval

Successivamente, si usa la funzione `map` per applicare il pruning e pulire tutte le stringhe del dataset

Ora abbiamo il dataset completo

La prima cosa da fare è dividere il dataset in due parti:

- una parte per addestrare il modello (**training** set)
- una parte per testare il modello (**test** set)

È importante non allenare mai il modello su tutto il dataset e poi testarlo sullo stesso dataset

Solitamente si prendono i $\frac{2}{3}$ del dataset il training e $\frac{1}{3}$ per il test

La funzione `train_test_split` prende due colonne (colonna dell'etichetta e colonna del testo) e richiede di specificare il parametro `random_state`.

Questo perché il dataset viene suddiviso in modo pseudocasuale.

Per rendere l'esperimento replicabile, si utilizza un valore costante per `random_state`.

Lo split restituisce `X_train` e `X_test`, ossia i dati di training e test, e le rispettive etichette.

Nella vita reale, non abbiamo le etichette nel test set.

Questa funzione di split, comunque, mantiene le proporzioni dei dati tra i due set creati.

Successivamente si passa all'indicizzazione dei documenti.

Non è molto diverso da ciò che è stato fatto in passato, ma la differenza è che prima applicavamo direttamente la TF-IDF sui documenti.

Ora, invece, dobbiamo prima calcolare le frequenze secche con `CountVectorizer` e poi passare il risultato alla funzione `fit_transform` di TF-IDF per ottenere un nuovo dataset.

Dopo questo passaggio, otteniamo la dimensione della matrice termini-documenti del training set.

La versione `MultinomialNB` di `naive_bayes` processa automaticamente le feature numeriche.

Il parametro importante nel costruttore di `MultinomialNB` è la variabile `alpha`, che rappresenta lo **stimatore di Laplace** (di default impostato a 1).

La funzione `fit` di `MultinomialNB` prende in input il modello TF-IDF e le etichette, costruendo così un modello di Naive Bayes.

Dopo aver costruito il modello, lo testiamo con il test set.

Prima, però, applichiamo una trasformazione con `CountVectorizer` e successivamente TF-IDF.

Infine, usiamo la funzione `predict` a cui passiamo i dati del test set.

In `fit` avevamo passato i dati con le etichette, mentre in `predict` non passiamo le etichette.

`predicted` restituirà le etichette previste. A questo punto, possiamo calcolare la media di quante volte `predicted` è uguale a `y_test`, cioè quante volte l'etichetta predetta corrisponde a quella reale

▼ Lezione 12 - 15/04/2024

Ripetizione Naive Bayes

Evaluating IR models

Valutare un sistema di IR consiste nel misurare **quanto bene risponde il sistema alle necessità dell'utente**

Fare ciò è difficile: dato lo stesso insieme di risultati è molto probabile che questi siano interpretati in modo diverso da diversi utenti.

Per affrontare questo problema sono state definite delle metriche in linea con le preferenze di un gruppo di utenti.

La valutazione dei documenti recuperati è una componente critica di un sistema IR.

La valutazione delle performance consiste nell'**associare una metrica quantitativa ai risultati prodotti da un sistema IR**

Le metriche devono essere associate direttamente con i risultati che l'utente ritiene rilevanti.

Di solito, **questo calcolo richiede di comparare i risultati ottenuti dal sistema con quelli suggeriti dall'utente**

Reference Collections

Le reference collections sono il metodo più usato per valutare un sistema IR.

Una reference collection è composta da:

- **D : insieme di documenti pre selezionati**
- **I : insieme di query**
- **Giudizi di rilevanza**: un insieme di valutazione che permette di indicare **se un documento è rilevante o meno per una determinata query**.

Questi giudizi sono per ogni coppia formata da una query appartenente ad I ed un documento appartenente a D .

Il giudizio ha valore 0 se il documento d non è rilevante per la query i o 1 se è rilevante.

Precision and Recall

Consideriamo:

- I : un **insieme di query**
- R : un **insieme di documenti rilevanti per I**
- A : i **documenti recuperati** dal sistema IR dopo aver analizzato la query.
- $R \cap A$: i **documenti rilevanti** che sono stati recuperati

Recall: è la porzione di documenti rilevanti (R) che sono stati recuperati rispetto al totale di documenti rilevanti.

$$Recall = \frac{R \cap A}{R}$$

Precision: è la porzione di documenti recuperati che sono rilevanti.

$$Precision = \frac{R \cap A}{A}$$

Consideriamo una reference collection e un insieme di query di test:

sia R_{q_1} l'insieme di documenti rilevanti per la query q_1

$$R_{q_1} = \{d_3, d_5, d_9, d_{25}, d_{39}, d_{44}, d_{56}, d_{71}, d_{89}, d_{123}\}$$

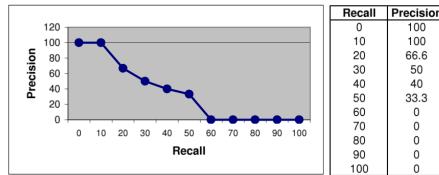
Consideriamo un nuovo algoritmo di IR che porta alle seguenti risposte alla query q_1 (i documenti rilevanti sono indicati con un punto)

01. d_{123} •	06. d_9 •	11. d_{38}
02. d_{84}	07. d_{511}	12. d_{48}
03. d_{56} •	08. d_{129}	13. d_{250}
04. d_6	09. d_{187}	14. d_{113}
05. d_8	10. d_{25} •	15. d_3 •

Osservando questa classifica notiamo che:

- il documento d_{123} è classificato come 1 ed è rilevante, questo documento corrisponde al 10% di tutti i documenti rilevanti e quindi possiamo dire che abbiamo una precision del 100% ad una recall del 10%
- il documento d_{56} è classificato come 3° ed è il seguente rilevante, a questo punto 2 doc su 3 sono rilevanti e 2 su 10 rilevanti sono stati osservati, quindi possiamo dire che abbiamo una precision del 66% ad una recall del 20%

Se procediamo con questa verifica della classifica possiamo plottare una curva di precision vs recall



In questo **esempio** la figura è stata calcolata sulle singole query.

Di solito però gli algoritmi di retrieve sono valutati lanciando diverse query distinte.

Per valutare le performance di recupero su N query, **effettuiamo la media della precision per ogni livello di recall come segue:**

$$\bar{P}(r_j) = \sum_{i=1}^N \frac{P_i(r_j)}{N}$$

dove:

- $\bar{P}(r_j)$ è la precision media al livello di recall r_j
- $P_i(r_j)$ è la precision al livello di recall r_j della i-esima query

La curva di precision-recall è usata normalmente per comparare le performance di diversi algoritmi IR.

Problemi della metrica precision-recall

- Una stima appropriata della massima recall di una query richiede una conoscenza profonda dei documenti della collezione
- In molte situazioni l'utilizzo di una singola metrica è più appropriato

La **curva recall - average precision** è lo standard utilizzato per valutare sistemi IR

Ci sono situazioni in cui vorremo valutare le performance su query individuali, il motivo è duplice:

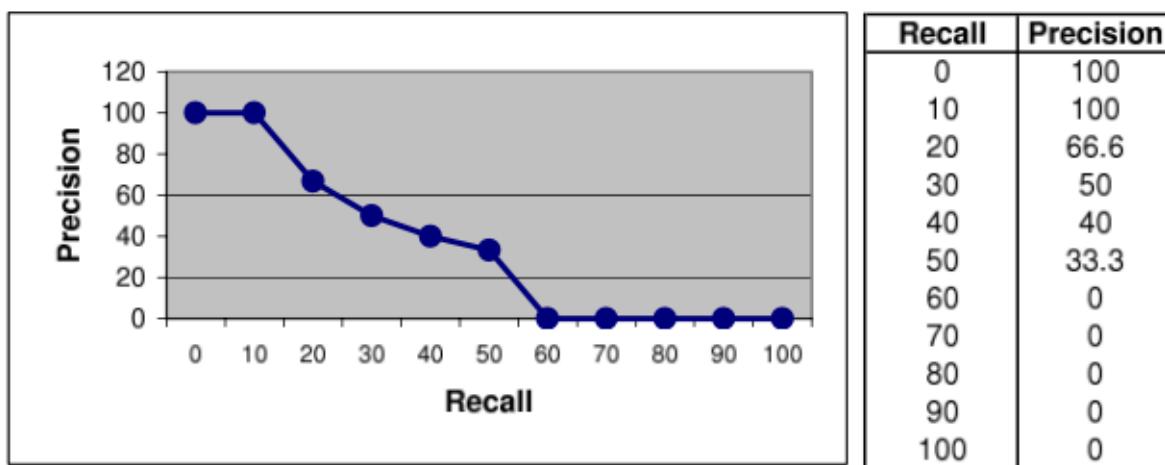
- fare la media della precision su molte query può nascondere delle anomalie.

- potremmo essere interessati nel sapere su quale query un algoritmo performa meglio di un altro.

MAP: Mean Average Precision

L'idea è di effettuare la media sulle curve di precision che si ottengono ogni volta che si osserva un nuovo documento, per documenti rilevanti che **NON** sono stati recuperati la precision è 0.

Prendiamo la curva precision-recall di prima per la query q_1



Quindi la MAP per q_1 è data da:

$$MAP_1 = \frac{1 + 0.66 + 0.5 + 0.4 + 0.33 + 0 + 0 + 0 + 0 + 0}{10} = 0.28$$

MRR Mean Reciprocal Rank

MRR è una buona metrica per quei casi dove siamo interessati alla prima risposta corretta come:

- QA system

Valutazione di Modelli di Machine Learning

È importante valutare le prestazioni di sistemi di IR e ML

Per valutare l'affidabilità e l'efficacia di un modello, si utilizzano alcune metriche che fungono da indici di rilevanza

Queste aiutano a comprendere come il modello si comporta nel recuperare e classificare correttamente i documenti

Vediamo alcune metriche:

- **Precision:** indica la frazione di documenti recuperati che sono rilevanti

$$(\text{Precision} = \frac{TP}{TP + FP})$$

- **Recall:** indica la frazione di documenti rilevanti che sono stati recuperati

$$(\text{Recall} = \frac{TP}{TP + FN})$$

- **Matrice di Confusione:** mostra il numero di documenti classificati correttamente e erroneamente:
 - **True Positives (TP):** documenti rilevanti correttamente identificati
 - **False Positives (FP):** documenti non rilevanti erroneamente identificati come rilevanti
 - **False Negatives (FN):** documenti rilevanti erroneamente identificati come irrilevanti
 - **True Negatives (TN):** documenti non rilevanti correttamente ignorati

Conclusioni

La valutazione dei modelli di machine learning è essenziale per garantire che le predizioni siano affidabili

L'uso di tecniche di preprocessing e la scelta dei parametri giusti sono cruciali per migliorare le performance del modello

Precision e recall sono metriche fondamentali per comprendere l'efficacia nel recuperare documenti rilevanti e per ottimizzare il modello di classificazione

La volta scorsa abbiamo visto le metriche usate per valutare i sistemi di IR (Information Retrieval).

Esistono diverse metriche, ciascuna delle quali presenta alcuni problemi.

Sistema di IR: data una query, restituisce un insieme di documenti.

Per valutare i sistemi ci sono due metriche principali:

PRECISION

Indica **quanti documenti rilevanti relativi alla query** sono stati restituiti rispetto al totale dei documenti indicizzati dal nostro sistema.

La precision viene calcolata solo quando c'è almeno un documento rilevante rispetto alla nostra query.

RECALL

Indica la **quantità di documenti rilevanti presenti nella risposta** in relazione al totale dei documenti rilevanti per la query.

In genere, la recall viene calcolata a livelli standard (da 0 a 10) e a quei livelli si calcola la precision, se ci sono documenti rilevanti nella risposta

Se calcoliamo la precision solo quando ci sono documenti rilevanti, può capitare che a certi livelli di recall non si abbiano dati.

Cosa si fa in questo caso? Si applica un'interpolazione lineare.

In questo modo otteniamo una nuova tabella che ci permette di fare un grafico (plot)

Confronto tra sistemi

Un aspetto importante è che possiamo confrontare due sistemi diversi oppure lo stesso sistema su query diverse.

Per fare questo, abbiamo bisogno di dati (da cui l'importanza dell'interpolazione).

Come calcoliamo la media per la precision?

Sommiamo i valori della precision ad un certo valore di recall e poi dividiamo per il numero di query.

MAP (Mean Average Precision)

La **MAP** ci dà un indice di precisione complessivo del sistema. Il risultato è un unico valore, cioè il valore medio della precision

Sommiamo tutti i valori di precisione e ne facciamo la media

MRR (Mean Reciprocal Rank)

La **MRR** fornisce un'idea generale delle prestazioni del sistema rispetto a una soglia.

Questa metrica indica quanti documenti rilevanti sono stati recuperati rispetto alla query, in base a una soglia.

Se il documento è di nostro interesse, ne prendiamo il reciproco e lo sommiamo.

In generale, sommiamo i reciproci delle posizioni dei documenti rilevanti e consideriamo solo quelli che sono inferiori alla soglia.

Questa è una metrica che valuta il **ranking**, ossia la classificazione, del sistema di IR.

METRICHE PER VALUTARE CLASSIFICATORI DI ML

I classificatori di Machine Learning (ML) ci permettono di classificare i dati in base a determinate condizioni e di assegnarli a specifiche classi.

Per valutare le prestazioni di questi sistemi, è importante assicurarsi che le metriche siano descrittive e rappresentative del comportamento del classificatore.

ACCURACY

L'accuracy è data dal **rapporto tra il numero di predizioni corrette e il numero totale di predizioni effettuate**

Tuttavia, questa metrica non è sempre completa.

Sebbene sia intuitiva, non fornisce informazioni affidabili in presenza di dataset sbilanciati

Esempio: nel caso delle nascite, se il dataset è fortemente sbilanciato perché ci sono molte più nascite maschili rispetto a quelle femminili, l'accuracy potrebbe risultare elevata anche se il classificatore non è realmente efficiente.

La qualità e l'affidabilità dei dati utilizzati per addestrare un classificatore sono quindi fondamentali per ottenere buoni risultati.

CONFIDENCE

Dati due modelli che, eventualmente, commettono gli stessi errori, la **confidence ci indica quale dei due è migliore e perché**

Questa metrica misura quanto un classificatore riesce a predire una classe con maggiore fiducia rispetto a un'altra.

La confidence misura la **percentuale di affidabilità della predizione**. In pratica, ci dice quanto siamo sicuri di una classificazione, basandosi sulle probabilità assegnate alle classi

Esempio: consideriamo la classificazione delle email in **spam** o **ham**.

La matrice che otteniamo mostra per ogni email le probabilità che sia spam o ham, con i valori da interpretare per righe

Le due colonne della matrice rappresentano:

- probabilità che l'email sia **ham**.
- probabilità che l'email sia **spam**.

Se il valore ottenuto è vicino a 1, possiamo essere quasi certi che la predizione sia corretta. Se il valore è prossimo a 0,5, le due probabilità (spam e ham) sono quasi uguali, quindi non possiamo dire con certezza se l'email appartiene a una classe o all'altra.

CONFUSION MATRIX

La **confusion matrix** (matrice di confusione) **NON è una metrica**, ma una tabella che ci aiuta a comprendere meglio le prestazioni del classificatore

Caratteristiche

Le dimensioni della matrice di confusione dipendono dal numero di classi: se abbiamo n classi, la matrice sarà di dimensione $n \times n$

- **Righe:** rappresentano le classi prodotte dal classificatore

- **Colonne:** rappresentano le classi assegnate dall'umano

Sulla **diagonale principale** si trovano i casi in cui il classificatore ha predetto correttamente (ad esempio, sia l'umano che il classificatore hanno classificato una mail come spam).

Sulla **diagonale secondaria**, invece, si trovano i casi in cui il classificatore ha commesso un errore.

Quando parliamo di "positivo" e "negativo", non indichiamo se il classificatore ha fatto bene o male, ma semplicemente due classi (ad esempio, **spam** e **ham**).

Nella matrice di confusione possiamo identificare quattro situazioni principali:

- **True Positive (TP):** la mail è ham e il classificatore ha predetto ham.
- **True Negative (TN):** la mail è spam e il classificatore ha predetto spam.
- **False Positive (FP):** la mail è spam, ma il classificatore l'ha segnalata come ham.
- **False Negative (FN):** l'umano ha classificato la mail come ham, ma il classificatore ha predetto spam

True Positive e **True Negative** si trovano sulla **diagonale principale**

False Positive e **False Negative** si trovano fuori dalla diagonale principale.

Metriche derivate dalla Confusion Matrix

Dalla matrice di confusione si possono calcolare diverse metriche:

- **Accuracy:** rappresenta la percentuale di predizioni corrette rispetto al totale. Questa metrica ci dice quante volte il classificatore ha predetto correttamente.
- **Error Rate:** rappresenta il tasso di errore, cioè la percentuale di predizioni errate. È il complemento dell'accuracy e si calcola come:

$$\text{Error Rate} = 1 - \text{Accuracy}$$

Come ottenere la Confusion Matrix con Python?

Con la libreria `sklearn`, è possibile utilizzare la funzione `metrics` per ottenere la confusion matrix. Bisogna passare come parametri le etichette di test (`y_test`) e le etichette predette dal modello (`predicted`).

Inoltre, `sklearn` fornisce anche la funzione `accuracy_score` per calcolare direttamente l'accuracy.

SENSITIVITY E SPECIFICITY

Queste metriche possono essere ricavate dalla matrice di confusione e aiutano a capire come un classificatore cerca di ridurre il numero di falsi positivi o di falsi negativi, a seconda di come è stato configurato per predire una classe rispetto all'altra.

Per analizzare meglio questo **trade off** introduciamo due nuove metriche: sensitivity e specificity.

Il **trade-off** tra le due metriche si analizza osservando come un classificatore bilancia l'identificazione dei veri positivi (minimizzando i falsi negativi) rispetto ai veri negativi (minimizzando i falsi positivi)

Vediamo più in dettaglio queste due metriche:

- **Sensitivity** (o Recall): valuta la proporzione di **True Positive** rispetto ai **False Negative**. Misura quanti dati di una determinata classe sono stati correttamente classificati.

Si calcola come:

D

- **Specificity**: nota anche come **True Negative Rate**, misura la proporzione di dati negativi che sono stati classificati correttamente. Si calcola come:

D

PRECISION E RECALL

Precision e Recall sono metriche molto simili a **Specificity** e **Sensitivity**, la differenza è che precision e recall si utilizzano per valutare le prestazioni di sistemi IR mentre specificity e sensitivity sono utilizzate per valutare le prestazioni di classificazione

- **Precision** (o Valore Predittivo Positivo):



- **Recall**



La differenza principale tra **Precision** e **Recall** è che la precision prende in considerazione i **False Positive**, mentre la recall considera i **False Negative**.

Questo perché la recall vuole capire quanti dei documenti classificati sono effettivamente corretti sul totale dei documenti recuperati.

Macro vs Micro Averaged Precision and Recall

- **Macro Average Precision:** calcola la precisione media aritmetica delle singole classi. Si sommano le precisioni di ogni classe e poi si fa la media.
- **Micro Average Precision:** è una media pesata, che tiene conto di tutti i valori e quindi dà maggiore peso alle classi più numerose. Si calcola come il rapporto tra il totale dei **True Positive** e il numero totale di predizioni.



La **micro average precision** assomiglia molto all'**accuracy**, poiché misura la percentuale di predizioni corrette sul totale delle predizioni.

- Macro averaged precision and recall sono la media di precisione e recall calcolata sulle categorie
- Micro averaged precision and recall sono uguali e corrispondono all'accuracy

Differenza tra Macro e Micro Average

- **Macro Average:** è la media delle metriche calcolate per ciascuna classe, indipendentemente dalla loro frequenza. È utile quando le classi hanno dimensioni simili.
- **Micro Average:** è una media pesata che considera la frequenza delle classi, ed è più indicata quando ci sono classi sbilanciate.

Uso delle metriche in Python

In Python, `sklearn` fornisce funzioni apposite per calcolare queste metriche.

Un'utilissima funzione è `classification_report`, che fornisce una panoramica completa delle metriche più importanti (precision, recall, f1-score, etc.) in un unico output.

Per valutare visivamente la bontà del modello, dobbiamo osservare la **matrice di confusione**: i valori sulla diagonale principale dovrebbero essere più alti rispetto agli altri, indicando che il modello ha classificato correttamente la maggior parte dei casi.

F-MEASURE (o F-SCORE)

È la combinazione di precision e recall, è la media armonica delle due, la media armonica è diversa dall'aritmetica poiché utilizza dei pesi.

$$\text{F-measure} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{2 \times TP}{2 \times TP + FP + FN}$$

In generale, il valore unico prodotto da metriche come l'**F-score** non sempre offre una visione completa delle prestazioni del classificatore.

Sebbene sia utile, l'F-score può nascondere comportamenti particolari del sistema.

Per questo motivo, non dobbiamo considerarlo come "oro colato".

Prima di tutto, è essenziale saper leggere il report delle varie metriche per capire cosa ognuna di esse indica e cosa potrebbe celare. In base ai risultati ottenuti, possiamo valutare se il classificatore soddisfa o meno i nostri obiettivi e preferenze.

ROC CURVE

Metriche come sensitivity, specificity ecc... valutano il modello utilizzando un singolo numero. Con la **roc curve** vedremo come questi modelli performano attraverso un grande range di condizioni.

È possibile che due modelli con simile accuracy possano aver ottenuto questo valore in modi diversi, per esempio minimizzando i falsi positivi o i falsi negativi.

La ROC curve (Receiver Operating Characteristic) esamina il tradeoff tra il rilevamento di **true positive** e l'evitamento di **false positive**.

La curva utilizza la proporzione tra **true positive** (asse verticale) e **false positive** (asse orizzontale)

Queste metriche sono equivalenti a sensitivity e 1-specificity, di conseguenza questo diagramma viene anche chiamato **sensitivity/specificity plot**.

Le predizioni dei classificatori sono ordinate grazie alla loro probabilità di essere classificati come positivi, in ordine decrescente (prima i valori più grandi)

Iniziando dall'origine, ogni predizione impatterà sulla curva in modo orizzontale (predizione non corretta) o in modo verticale (predizione corretta)

La **ROC Curve** (Receiver Operating Characteristic) prende in considerazione il tasso di **True Positive** e **False Positive**.

Attraverso questo grafico, è possibile ottenere diverse informazioni sul comportamento del classificatore.

Se la curva è una retta diagonale (ovvero, una retta con pendenza 45 gradi), significa che il sistema non è efficace, poiché equivale a tirare una moneta ogni volta (con il 50% di probabilità di predire correttamente o sbagliare).

Questo indica un classificatore casuale e quindi poco utile.

Il grafico ROC è anche noto come grafico **Specificity** (False Positive Rate) e **Sensitivity** (True Positive Rate).

Con la **ROC Curve**, possiamo confrontare due classificatori differenti oppure valutare come si comporta lo stesso classificatore al variare delle query.

AUC (Area Under the Curve)

L'**AUC** rappresenta l'area sotto la curva ROC

È un indice numerico che permette di valutare le prestazioni di un classificatore senza dover osservare direttamente il grafico

Tuttavia, possono sorgere problemi poiché curve diverse possono avere la stessa area sottostante

Idealmente, una curva ROC che cresce rapidamente indica che il classificatore ha un buon rapporto tra true positive e false positive.

È preferibile che il classificatore massimizzi i **True Positive** su un numero relativamente basso di predizioni.

Anche se si lavora con un dataset più grande, il ragionamento rimane valido.

▼ Lezione 14 - 22/04/2024

ESTIMATING FUTURE PERFORMANCE

Quando costruiamo un modello di machine learning, il problema principale è valutare quanto bene il modello riesca a predire su dati che non ha mai visto

Normalmente, quello che facciamo è prendere l'intero dataset, costruire il modello su tutti i dati e valutare come il modello si comporta su quei dati

Ciò ci dice quanto bene il modello ha **fittato** i dati, ma non ci dà informazioni su come il modello predice su dati nuovi o mai visti

Per risolvere questo problema, dividiamo il dataset in due parti:

- una parte per il **training**
- una parte per il **test**.

Questo processo è noto come **holdout**.

Soltanente, dividiamo il dataset in circa 2/3 per il **training set** e 1/3 per il **test set**.

Valutare le performance del modello in questo modo ha delle limitazioni.

Se siamo fortunati e il **test set** assomiglia molto al **training set**, il modello può sembrare performante.

Tuttavia, se il **test set** è troppo diverso dal **training set**, il modello potrebbe avere difficoltà a generalizzare e predire correttamente.

Un altro problema si presenta quando abbiamo un dataset molto piccolo.

In questi casi, la divisione in training e test set potrebbe non essere sufficiente per dare un'idea accurata delle prestazioni del modello.

Miglioramento: aggiunta del validation set

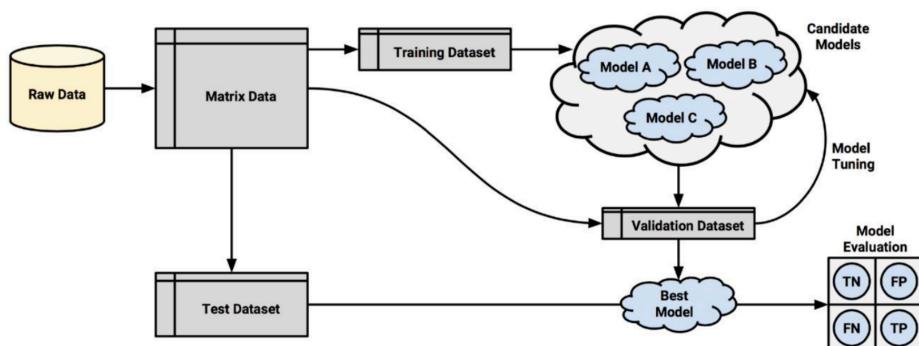
Per affrontare questo problema, possiamo introdurre un **validation set**.

In questo caso, dividiamo il dataset in tre parti:

1. **Training set** (70%): usato per addestrare il modello.
2. **Validation set** (20%): usato per validare il modello durante la fase di tuning dei parametri.
3. **Test set** (10%): usato solo alla fine per valutare le prestazioni finali del modello.

Il **validation set** ci permette di fare un'ulteriore valutazione del modello, aggiungendo un passaggio intermedio.

Questo processo aiuta a evitare il problema di ottimizzare il modello esclusivamente sui dati del training set, migliorando la capacità del modello di generalizzare su dati nuovi.



Il **test set** viene utilizzato solo alla fine del processo, cioè dopo aver scelto il modello definitivo. Prima, si addestrano vari modelli e si utilizza il **validation set** per capire quale modello funziona meglio

Perché non confrontiamo i modelli direttamente sul test set?

Perché il test set deve simulare dati mai visti, proprio come avverrà nella vita reale.

Il suo scopo è valutare la capacità del modello di generalizzare, quindi usarlo per confrontare modelli durante la fase di selezione comprometterebbe questa valutazione.

Problemi con il campionamento casuale

Supponiamo di avere un dataset composto dal 20% di email **spam** e dall'80% di email **ham**.

Se facciamo una suddivisione casuale tra **training set** e **test set**, potrebbe accadere che il test set contenga solo email spam o solo email ham, o che il training set sia completamente sbilanciato.

Questo sarebbe un problema, poiché il test set deve riflettere le proporzioni originali del dataset, per garantire una valutazione realistica del modello.

Il **training set**, invece, può essere sbilanciato, anche se è preferibile che sia bilanciato per migliorare l'addestramento del modello.

Per fortuna, l'API Python `train_test_split` di `scikit-learn` permette di fare uno **split proporzionato** in base alle categorie, garantendo che il **training set** e il **test set** mantengano le stesse proporzioni

Quando creiamo i set di training e test, è quindi importante fare una suddivisione **stratificata** anziché un campionamento casuale puro

Problemi con dataset piccoli

Un problema comune quando si lavora con dataset piccoli è che il modello potrebbe adattarsi bene al training set, ma il test set, essendo troppo piccolo, non fornirebbe una validazione significativa

Soluzione: Cross-Validation

Per risolvere questo problema, utilizziamo la tecnica della **cross-validation**.

Con la cross-validation, dividiamo il dataset in **k** sottoinsiemi (o "fold").

A ogni iterazione, utilizziamo uno dei sottoinsiemi come **test set**, mentre gli altri **k-1** sottoinsiemi vengono utilizzati come **training set**.

Dopodiché, cambiamo il sottoinsieme di test e ripetiamo il processo fino a utilizzare tutti i sottoinsiemi sia per il training che per il testing

Un valore tipico di **k** è 10, quindi si parla di **10-fold cross-validation**.

In questo modo, costruiamo e validiamo 10 modelli differenti, e testiamo il modello su tutto il dataset.

Questo ci permette di validare il modello su dati differenti, poiché ogni sottoinsieme viene utilizzato sia per il training che per il testing.

Questa strategia è molto usata, ma lo **svantaggio** principale è il **tempo richiesto**, poiché bisogna costruire più modelli (uno per ogni iterazione)

Cross-Validation con validation set

La stessa strategia può essere applicata anche quando si vuole includere un **validation set**.

In questo caso, si riserva un'ulteriore porzione del dataset per la validazione, mentre la **cross-validation** viene eseguita sui dati rimanenti.

In Python, questo processo è implementato in `scikit-learn`, che fornisce diverse funzioni per eseguire la cross-validation.

Tra queste, ci sono anche funzioni specifiche per dataset **temporali**.

Cross-Validation su dati temporali

La cross-validation funziona bene quando i dati nei vari fold non hanno un ordine temporale, ossia quando i dati sono **indipendenti tra loro**.

In un contesto temporale, però, l'ordine dei dati è importante.

Ad **esempio**, se abbiamo un dataset di segnalazioni di bug su GitHub, non possiamo usare i dati futuri per addestrare il modello e poi testarlo su dati del passato.

Questo creerebbe un errore nella valutazione, poiché useremmo informazioni che non sarebbero disponibili in uno scenario reale.

Per gestire dataset temporali, esistono funzioni specifiche che suddividono il dataset incrementando il **training set** nel tempo, facendo sempre in modo che il **test set** contenga dati **futuri** rispetto al training.

Codice Python per Cross Validation

Con la funzione `cross_val_score` possiamo calcolare una sola metrica, passando come parametro `scoring=nome_metrica`, in questo modo avremo il calcolo di una metrica su tutti i

modelli creati.

Per avere l'accuracy completa dovremmo poi farne la media.

`scoring` accetta anche un dizionario come parametro, utile nel caso in cui si volessero calcolare più metriche.

Se al parametro `n_jobs` passiamo un valore positivo, questo farà l'esecuzione su più thread

Importanza delle feature nei modelli predittivi

Quando costruiamo un modello predittivo, oltre a valutare metriche come l'accuracy, è importante comprendere **quali variabili indipendenti** (le **feature**) contribuiscono maggiormente alla predizione

Questo è essenziale per interpretare il comportamento del modello e per fare inferenze utili sui dati.

Spiegare le feature in un modello

Con l'algoritmo di **Naive Bayes**, possiamo accedere alla probabilità associata a ciascuna feature (variabile indipendente)

In particolare, la funzione `feature_log_probability` ci consente di osservare la probabilità logaritmica associata a ciascuna feature per ciascuna classe, ad esempio per **ham** e **spam** in un modello di classificazione di email.

Selezione delle feature più rilevanti

Spesso ci interessa trovare le **feature più significative** per una classe.

Per fare ciò, possiamo seguire i seguenti passaggi:

1. **Accesso alle probabilità logaritmiche**: utilizziamo `feature_log_probability`, che ci fornisce un array con le probabilità logaritmiche per ciascuna feature.
2. **Uso di argsort** : per identificare le feature più rilevanti, utilizziamo la funzione `argsort` di `numpy`, che ordina gli indici di un vettore come se fosse ordinato. In questo modo possiamo trovare facilmente le **feature più rilevanti** sia per "ham" che per "spam", in particolare:
 - **gli indici più alti** rappresentano le feature con **maggior probabilità di appartenere** a una classe.
 - **gli indici più bassi** rappresentano le **feature con minore probabilità**.

3. **Estrazione delle top feature:** dopo aver ottenuto gli indici ordinati, possiamo usare `np.take`, una funzione di numpy che consente di estrarre gli elementi da un array o dataframe sulla base di specifici indici. Questo ci permette di estrarre le **top 20 feature** in termini di probabilità per ciascuna classe.

In pratica, questo processo ci consente di spiegare quali sono le feature che hanno il maggiore impatto nel determinare la classificazione di un'email come "ham" o "spam", o di qualsiasi altra classe nel contesto del nostro modello.

N-GRAM MODELS FOR TEXT PREDICTION

Ora costruiamo il primo modello generativo probabilistico, considerato il precursore delle reti neurali

Applicazioni:

- traduzione automatica
- correzione ortografica
- riconoscimento vocale
- riassunto automatico

La prima assunzione da fare è che **le frasi seguano sequenze regolari di parole**, quindi possiamo predire quali saranno le parole successive

Il concetto di **modello di linguaggio** si applica anche ai modelli neurali.

Vogliamo calcolare **la probabilità di una parola successiva data una sequenza di parole**.

Un qualsiasi modello che faccia questo, sia che si basi sul teorema di Bayes, sia che sia un modello neurale, viene definito un **modello di linguaggio** (GPT è un esempio di questo).

Se ho una sequenza di parole, voglio calcolare la probabilità dell'intera sequenza.

Posso usare il **chaining delle probabilità**, applicando il teorema di Bayes:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

Questa formula può essere estesa a più variabili: la probabilità della parola A, poi la probabilità di B condizionata su A, la probabilità di C condizionata su A e B, e così via

Quindi, per una sequenza di parole, otteniamo una catena di probabilità condizionate:

- **probabilità della prima parola**
- **probabilità della seconda parola condizionata dalla prima**
- **probabilità della terza parola condizionata dalle due precedenti**

Tutto questo si traduce in un prodotto di probabilità, dove il primo elemento non è condizionato

Per calcolare questi valori, utilizziamo la **likelihood**, ovvero contiamo **quante volte una certa sequenza di parole compare nel set di addestramento**

Il problema del conteggio delle sequenze

Il problema nasce dal fatto che dovremmo contare tutte le possibili sequenze di parole.

La soluzione che viene adottata è basata sull'**assunzione di Markov**

La **catena di Markov** è un modello probabilistico che descrive una sequenza di eventi in cui **la probabilità di ogni evento dipende solo dallo stato attuale e NON dagli eventi passati**

Poiché ogni parola è considerata un evento, non è necessario tener conto di tutte le parole precedenti nella catena, ma possiamo limitarci a **n** parole precedenti.

Da qui deriva il concetto di **modello basato su n-grammi**

Modelli basati su n-grammi

Se utilizziamo un modello basato su **bi-grammi**, ad **esempio**, consideriamo solo la parola precedente per predire la successiva

In generale, **possiamo approssimare la sequenza riducendo la finestra di osservazione a n parole precedenti**

Con gli **uni-grammi**, prediciamo ogni parola in base alla sua probabilità di apparire nel dataset, **senza considerare il contesto** delle parole precedenti

Con i **bi-grammi**, prediciamo una parola in base alla parola immediatamente precedente.

Con i **tri-grammi** e n-grammi di ordine superiore, consideriamo n-1 parole precedenti per fare la predizione.

Stimare le probabilità degli n-grammi

Per stimare le probabilità degli **n-grammi**, usiamo la **maximum likelihood estimate (MLE)**, che stima quante volte la parola w_i è seguita dalla parola w_{i-1} , rispetto a quante volte la parola w_{i-1} appare nel dataset.

In altre parole, calcoliamo la probabilità condizionata delle parole successive sulla base delle precedenti

Calcolo pratico delle probabilità

Il calcolo di queste **probabilità** viene fatto in maniera **logaritmica**

La **moltiplicazione** delle **probabilità**, infatti, **si può trasformare in una somma** grazie all'uso dei logaritmi, il che rende il modello più efficiente da un punto di vista computazionale.

Il primo modello di n-grammi addestrato su Wikipedia è stato rilasciato da Google nel 2006.

Valutazione dei modelli di linguaggio

Come possiamo valutare l'efficacia di questi modelli?

In maniera simile ai classificatori, possiamo stimare le probabilità sul **training set** e usare un ulteriore dataset per il **test**.

La fase di **validation** non è molto utilizzata nei modelli di linguaggio, ma si può impiegare per valutare se un modello a 2-grams, 3-grams, ecc. funziona meglio di un altro.

Perplexity

Per **valutare un modello di linguaggio**, possiamo utilizzare una metrica chiamata **perplessità**.

La **perplessità misura quanto un modello è "incerto" nelle sue predizioni**: data una frase, la perplessità ci indica quante parole potrebbero seguire quella frase

Formalmente, la **perplexity** è la radice n-esima di 1 diviso la probabilità della sequenza di parole predette dal modello

La radice serve per ridurre l'ampiezza del valore finale, che altrimenti sarebbe troppo grande. In generale, **più bassa è la perplessità, meglio è**: un valore basso indica che il modello è più sicuro delle sue predizioni

Problema degli eventi mai visti

Un problema che può insorgere è la presenza di eventi che **NON si sono mai verificati nel dataset di addestramento**

In questi casi, le **probabilità di tali eventi sarebbe pari a zero**, causando problemi nel calcolo complessivo delle probabilità (un prodotto di zeri dà zero)

Per risolvere questo problema, si utilizza la **correzione di Laplace**, sostituendo gli 0 con piccoli valori prossimi allo zero ma diversi da zero.

Applichiamo il modello N-Grams utilizzando la libreria `nltk`, che contiene già un modello di linguaggio predefinito:

1. **Importazione dei bi-grammi:** importiamo la funzione `bigrams` dalla libreria `nltk.util` e creiamo i bi-grammi, ossia coppie di parole consecutive.
2. **Padding delle frasi:** importiamo una funzione che effettua il "padding" delle frasi (come `pad_both_ends`). Il padding è necessario per segnalare dove una frase inizia e finisce, aggiungendo simboli specifici all'inizio e alla fine della sequenza.
3. **Costruzione degli everygrams:** creiamo gli everygrams, cioè tutte le sequenze di parole di lunghezza variabile (ad **esempio**, uni-grammi, bi-grammi, tri-grammi). In questo caso, generiamo gli everygrams di lunghezza 3.
4. **Uso di `padded_everygram_pipeline`:** utilizziamo la funzione `padded_everygram_pipeline`, che restituisce due oggetti: uno contenente i n-grammi con padding, e un altro che contiene il vocabolario.

5. **Applicazione su un dataset reale:** per mostrare come applicare il modello N-Gram su un dataset reale, è stato preso un dataset con i tweet di Donald Trump. La prima operazione è una funzione di **clean up**, che pulisce i tweet utilizzando espressioni regolari. Questa funzione rimuove URL, link alle immagini e caratteri speciali.
6. **Tokenizzazione dei tweet:** dopo la pulizia, i tweet vengono tokenizzati, ovvero divisi in singole parole (token).
7. **Generazione dei tri-grammi:** dopo la tokenizzazione, si applicano i tri-grammi, ovvero gruppi di tre parole consecutive.
8. **Uso della classe MLE:** si utilizza la classe `MLE` (Maximum Likelihood Estimate) per stimare le probabilità degli n-grammi. Il numero di n-grammi da considerare è passato come parametro alla classe.
9. **Esempio di generazione:** come **esempio**, il professore ha fatto generare due parole successive alla frase "i will", utilizzando il modello MLE addestrato sui tri-grammi

▼ Lezione 15 - 29/04/2024

Nella lezione precedente abbiamo discusso dell'importanza delle feature, e come con modelli come **Naive Bayes** o **Decision Tree** sia semplice determinare quali feature influenzano di più le predizioni.

Tuttavia, questo diventa molto più complesso con modelli come le **reti neurali**, che sono meno interpretabili

Oggi ci concentreremo su come **manipolare il dataset** per migliorare le predizioni, applicando tecniche di **feature selection** per eliminare variabili ridondanti o inutili e mantenere solo quelle più significative

Questo approccio va oltre la semplice selezione delle parole più importanti, come abbiamo visto nelle lezioni precedenti

Miglioramento tramite selezione delle feature

L'idea alla base è selezionare solo le features che hanno un impatto positivo sulle predizioni e **rimuovere quelle che non hanno variabilità o forniscono informazioni ridondanti**

Alcuni modelli, come i **decision tree**, implementano meccanismi di selezione delle feature internamente.

In altri casi, come per i modelli che **non** fanno feature selection in modo automatico, possiamo utilizzare diverse tecniche manuali per ottenere lo stesso risultato

Le tecniche principali per la **selezione delle feature** includono:

- **Univariate Feature Selection:** questa tecnica seleziona le feature che **correlano bene con la variabile dipendente** e che **non sono fortemente correlate tra loro**. Si basa su test statistici univariati per trovare le feature più rilevanti.
- **Removing Features with Low Variance:** feature che presentano una varianza molto bassa non forniscono informazioni significative per la predizione, poiché hanno lo stesso valore in quasi tutte le istanze del dataset. Queste feature vengono eliminate.
- **Recursive Feature Elimination (RFE):** questa tecnica rimuove le feature in modo **ricorsivo**, eliminando quelle meno significative finché non si raggiunge un set minimo di feature che porta alle migliori predizioni.

Esempio:

Immaginiamo di avere una variabile dipendente che classifica le email come **spam** o **ham**, e tre variabili indipendenti x_1 , x_2 , e x_3 :

- Se x_1 ha sempre lo stesso valore, sarà eliminata perché non aggiunge valore predittivo.
- Se x_2 e x_3 sono altamente correlate con la variabile dipendente, ma sono anche correlate tra loro, allora possiamo mantenere solo una delle due perché portano quasi lo stesso contributo informativo.

Selezione automatica delle feature

Queste operazioni di selezione delle feature possono essere fatte in modo automatico con vari strumenti e metodi statistici.

In `scikit-learn`, esistono diversi strumenti per fare **feature selection**.

Ad **esempio**, il metodo `SelectKBest` utilizza il **test del chi-quadrato** χ^2 per confrontare la distribuzione delle variabili rispetto alla variabile dipendente e determina quali eliminare

In particolare, il **test del χ^2** verifica se la proporzione di due variabili differisce significativamente tra diverse categorie

Se due variabili hanno distribuzioni simili su due categorie, possiamo eliminare una delle due.

Per variabili numeriche, usiamo il test `f_classif`, che verifica **se due variabili hanno distribuzioni simili** e, in caso affermativo, ne elimina una.

Queste tecniche sono progettate per selezionare variabili che **correlano bene con la variabile target** e per evitare **ridondanza** tra le feature

È importante sottolineare che **la selezione delle feature deve essere fatta esclusivamente sul training set** perché il test set rimane sconosciuto fino alla fase finale per simulare la predizione su dati mai visti nella vita reale.

Ribilanciamento del Training Set

Quando si lavora con modelli di machine learning, è comune affrontare il problema di un **dataset di addestramento sbilanciato**, dove una categoria è molto più rappresentata rispetto all'altra.

Questo squilibrio può portare a risultati insoddisfacenti, perché il modello tende a imparare bene la classe maggioritaria, trascurando quella minoritaria.

Di conseguenza, in fase di test, il modello potrebbe ottenere alte performance generali (ad esempio, in termini di **accuratezza**), ma scarse performance sulle classi meno rappresentate (in particolare in termini di **recall**)

Esempio di dataset sbilanciato:

Immaginiamo di avere un training set con **10 spam** e **2000 ham**.

Durante il test, potremmo avere **10 spam** e **1000 ham**.

Un modello sbilanciato potrebbe predire correttamente quasi tutti gli **ham**, ma predire correttamente solo **9 spam su 10**. Sebbene l'accuratezza del modello sembri alta, la **recall** sugli spam è molto bassa, poiché solo 1 su 10 spam è stato effettivamente classificato correttamente.

Questo squilibrio può portare a un modello inefficiente, che di fatto non risolve il problema, perché potrebbe comportarsi addirittura peggio di un modello che predice costantemente solo la classe maggioritaria

Come bilanciare il training set?

Per migliorare le prestazioni del modello, è essenziale bilanciare il **training set** affinché il modello sia in grado di apprendere adeguatamente entrambe le classi (sia la maggioritaria che la minoritaria).

Tuttavia, il **test set** può rimanere sbilanciato per riflettere accuratamente i dati reali.

Ci sono due tecniche principali per bilanciare il training set

Undersampling

Consiste nel **ridurre la dimensione** della classe maggioritaria in modo che entrambe le classi abbiano lo stesso numero di sample.

Esempio: Se abbiamo **1200 ham** e **400 spam**, possiamo ridurre il numero degli **ham** selezionando casualmente solo **400** esempi, così da bilanciare le due classi.

Svantaggi: Il principale svantaggio è che si perdono molti dati, e se il dataset è già piccolo, questo approccio potrebbe ridurre eccessivamente la quantità di informazioni disponibili per l'addestramento.

Oversampling

È l'approccio opposto: **consiste nell'aumentare** la dimensione della classe minoritaria, generando più esempi da essa.

Esistono due principali tecniche di oversampling:

- **Replica degli esempi esistenti:** si moltiplicano gli esempi della classe minoritaria fino a raggiungere la stessa dimensione della classe maggioritaria

Svantaggi: Il problema di questa tecnica è che i classificatori possono rendersi conto che alcuni esempi vengono ripetuti più volte, e questo potrebbe non aggiungere nuove informazioni utili al modello.

- **Generazione di campioni artificiali:** invece di replicare gli esempi esistenti, si utilizzano tecniche come **SMOTE** (Synthetic Minority Over-sampling Technique) per generare nuovi esempi simili ma non identici agli esempi della classe minoritaria.

Vantaggi: questa tecnica evita la ripetizione esatta degli stessi esempi, migliorando la generalizzazione del modello.

Svantaggi: creare campioni artificiali potrebbe introdurre rumore nel dataset, e in alcuni contesti (specialmente in fase di test) potrebbe non funzionare in modo ottimale.

MODELLI DI ML ALTERNATIVI

Un classificatore molto semplice è il **K-Nearest Neighbor (KNN)**

Se un documento ha nel suo intorno più documenti di spam che di ham, allora sarà classificato come spam. Si **basa quindi sulla distanza tra i documenti**

Altri esempi:

- **Decision Tree:** un albero decisionale divide i dati in base a una regola, ad esempio: se una variabile è maggiore di un certo valore, si posiziona il punto a destra o a sinistra del nodo.
- **Ensemble Classifier:** a volte abbiamo dataset complessi che un singolo classificatore, soprattutto se superficiale (shallow), non riesce a spiegare bene. In questi casi si costruiscono più alberi decisionali su sottosinsiemi del dataset. In fase di test, la classificazione viene eseguita da tutti questi classificatori e la decisione finale viene presa a maggioranza.

Si costruisce quindi una "foresta" di alberi

Le

Random Forests sono tra i modelli più efficaci nella classificazione di testi, insieme alle **Support Vector Machines (SVM)**

SUPPORT VECTOR MACHINE (SVM)

Una **SVM stima una superficie (iperpiano) per separare i punti in due categorie**

Se ci sono più categorie, la SVM costruisce più piani.

Con due variabili, il piano sarà una retta che separa i punti nel miglior modo possibile.

Con molte dimensioni, il modello è efficace, ma può essere computazionalmente impegnativo.

Quale iperpiano scegliamo?

Si cerca l'**iperpiano** che **massimizza la distanza** tra i punti delle due categorie

Dato un vettore (iperpiano), si identificano i punti più vicini da un lato e dall'altro, tracciando **vettori paralleli** chiamati **support vector**, che creano un "**margine**" rispetto all'iperpiano.

Tuttavia, non è sempre possibile identificare un iperpiano che separi perfettamente due categorie. Quando lo spazio è **convesso**, è possibile trovare un iperpiano separatore, un concetto importante anche per le reti neurali.

Se lo spazio NON è perfettamente separabile, alcuni punti possono finire dal lato sbagliato dell'iperpiano. In questi casi, si aggiunge un **fattore di penalizzazione** per ogni punto che si trova dalla parte errata.

L'obiettivo generale è creare un "materasso" (margin) più grande possibile, **massimizzando la distanza tra i support vector**.

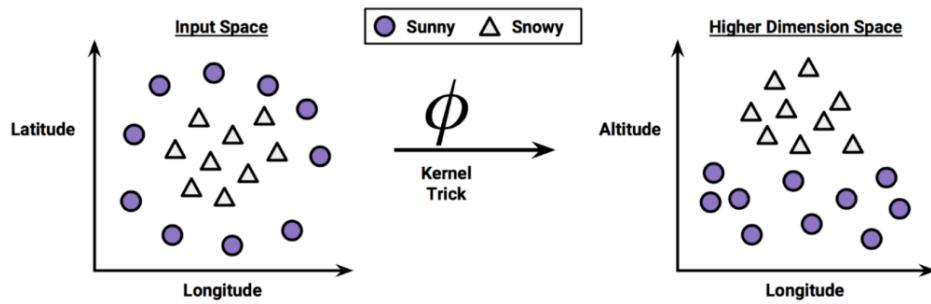
Se però alcuni punti finiscono nel margine sbagliato, vengono introdotte delle variabili di **slack**, che rappresentano le penalizzazioni. Nella funzione di ottimizzazione, per massimizzare il margin, si inseriscono penalizzazioni per ogni punto (indicate con ζ)

Ricapitoliamo

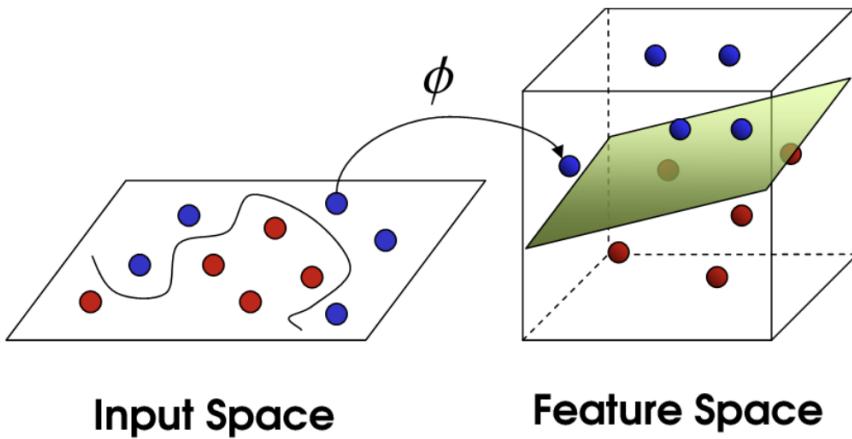
La SVM cerca di creare un margin il più grande possibile per distanziare le due categorie. Tuttavia, per ogni punto che finisce nel lato sbagliato, viene aggiunta una penalizzazione.

Inoltre, possiamo trasformare il nostro spazio applicando una **funzione di kernel**. Questa funzione aggiunge nuove dimensioni, ottenute dalle dimensioni esistenti, permettendo di creare un iperpiano che separa meglio le categorie.

Esempio: se abbiamo due dimensioni, come longitudine e latitudine, potremmo aggiungere una terza dimensione come altitudine per ottenere una separazione più efficace.



In 3D:



Kernel Transformation

Una kernel transformation prende uno spazio a n dimensioni e lo mappa in un nuovo spazio a più dimensioni tramite una funzione di kernel

Questa funzione definisce come calcolare il *dot product* (prodotto scalare) nel nuovo spazio

Pro e contro del modello SVM

I **vantaggi** di un modello SVM includono il fatto che **funziona bene** sia per problemi **numerici** che per problemi di **predizione**. È efficace in presenza di grandi quantità di

dati, **NON soffre di overfitting** ed è più semplice da utilizzare rispetto alle reti neurali.

Tuttavia, **va calibrato correttamente**: è necessario scegliere il kernel giusto, e ogni kernel ha dei coefficienti che devono essere stimati (anche il fattore di penalizzazione va determinato)

Siccome il modello è complesso, **il processo di addestramento può risultare costoso in termini di risorse computazionali**. Inoltre, **NON è semplice interpretare** un modello di questo tipo.

Funzione di kernel

Date due variabili \vec{x}_i e \vec{x}_j , la funzione di kernel prende questi due vettori e li combina tramite il prodotto scalare, restituendo un singolo numero.

Questa funzione permette di eseguire il calcolo del **prodotto scalare in uno spazio più grande rispetto a quello che osserviamo solitamente con due sole variabili**.

Ad **esempio**, immaginiamo una funzione di kernel che prende due vettori e li trasforma in:

►

Calcoliamo il kernel su questi due vettori:

►

Otteniamo una certa equazione che può essere riscritta come il *dot product* di due vettori. Possiamo quindi considerare quei vettori come il prodotto di due vettori a sei dimensioni, senza dover conoscere esattamente come è fatto questo spazio.

Tipi di kernel

- **Kernel Lineare:** il kernel più semplice che possiamo usare è: $K(x_i, x_j) = x_i \cdot x_j$
- **Polynomial Kernel:** qui il kernel è: $(x_i \cdot x_j + 1)^d$. Il parametro d deve essere stimato

- **Sigmoid Kernel:** basato sulla tangente iperbolica, per questo kernel è necessario stimare i parametri k e δ .
- **Gaussian Kernel:** è il kernel più versatile, in cui il valore di σ va stimato

Molto spesso il kernel lineare risulta particolarmente efficace con dati testuali, perché in questi casi ci troviamo di fronte a matrici fortemente sparse.

Come stabilire i valori dei parametri?

Un modo per determinare i valori ottimali dei parametri è utilizzare una procedura di calibrazione chiamata **Grid Search**

Esistono due varianti principali:

- **Random:** si provano valori casuali per i parametri.
- **Exhaustive:** si definisce una serie predefinita di valori da testare.

Come applicare Grid Search?

1. Con `get_params().keys()`, otteniamo la lista di tutti i parametri che possiamo calibrare.
2. Si specifica un dizionario con n chiavi, dove ciascuna chiave rappresenta un parametro da calibrare, associato a una lista di valori da provare.
3. A `GridSearchCV` si passa l'istanza del modello SVC, i parametri e il numero di *job* (ad esempio, `n_jobs=-1`) per eseguire il calcolo in modalità multithreading

Durante l'esecuzione di Grid Search, viene fatta una **cross-validation**, e per ogni *fold* vengono testate tutte le possibili combinazioni dei parametri specificati nel dizionario

Il Grid Search crea automaticamente un *validation set* per ogni *fold*.

Alla fine del processo, possiamo vedere i risultati ottenuti:

- Con `best_score_` otteniamo il miglior punteggio raggiunto, che di default è misurato in termini di *accuracy*.
- Con `best_estimator_` recuperiamo il miglior modello possibile generato dal Grid Search.

Una volta ottenuto il miglior modello, possiamo usarlo per fare le predizioni

▼ Lezione 16 - 02/05/2024

CREATING PIPELINE

Vedremo un modo per incapsulare l'intera pipeline in un unico oggetto, con due obiettivi principali:

- calibrare l'intero processo, non solo gli iperparametri
- gestire la trasformazione dei dati dall'input fino alla creazione del modello in un unico step

Come creiamo una pipeline per incapsulare tutto il processo di analisi dei dati?

Vediamo come farlo con un notebook.

Per creare una pipeline, dobbiamo istanziare un processo:

1. importiamo la classe `Pipeline` da `sklearn`.
2. istanziamo il modello SVM.
3. creiamo un oggetto `Pipeline`, costituito da una lista di tuple. Ogni tupla è formata da una chiave e un valore, dove la chiave rappresenta lo *stage* della pipeline e il valore è l'oggetto di `sklearn` che effettua la trasformazione.

Con dataset grandi questa roba si siede tranquillamente

Dopo aver creato la pipeline, procediamo alla calibrazione definendo una lista di parametri

I nomi dei parametri seguono una convenzione particolare:

- ogni parametro deve essere preceduto dal nome dello *stage* (cioè la chiave della tupla passata all'oggetto `Pipeline`), seguito da due underscore (`_`), e poi il nome del parametro specifico dell'oggetto `sklearn`.

Ora, invece di passare il modello `SV M` a `grid_search`, passiamo l'intera pipeline.

Successivamente, passiamo a `grid_search` i dati `x_train` e `y_train`, che hanno subito solo il preprocessing ma non sono ancora stati utilizzati per l'addestramento.

Dopo il testing, otteniamo il miglior modello, che non sarà solo una SVM, ma un'intera pipeline.

ADDING A CUSTOMIZED TRANSFORMER TO THE PIPELINE

Anche le scelte fatte nella **pulizia del testo possono essere calibrate**

Non è detto, ad esempio, che lo **stemming** o la **rimozione delle stop words** siano sempre utili

Per questo motivo, possiamo implementare una classe che estende i trasformatori di `sklearn`

Un trasformatore personalizzato deve implementare i seguenti metodi:

- **fit**: adatta i dati al modello.
- **transform**: applica la trasformazione ai dati rispetto al modello. Ad **esempio**, se ho costruito un modello TF-IDF con un determinato vocabolario, `transform` prende i dati del test set e li proietta sul modello, utilizzando solo le parole presenti nel vocabolario del modello.
- **fit_transform**: effettua sia il fit che il transform

È importante notare che **transform** non crea il modello, ma parte dal vocabolario esistente e, per ogni parola presente nei documenti del test set, la mappa al vocabolario già costruito.

COME CREIAMO UN OGGETTO TRANSFORMER PERSONALIZZATO

Per creare un oggetto transformer personalizzato, dobbiamo seguire alcuni passaggi chiave.

In particolare, dobbiamo creare una classe che:

1. erediti `BaseEstimator` e fornisca i metodi `set_params` e `get_params`.
2. erediti `TransformerMixin`

Quindi, la classe avrà un'**ereditarietà multipla**.

La classe deve implementare tre metodi principali:

- **costruttore**: il costruttore non esegue operazioni complesse, ma si limita a ricevere dei parametri e a creare gli attributi della classe, che saranno gli iperparametri per il processo di pulizia del testo. Ad **esempio**:
 - `stop`: indica se effettuare la rimozione delle stop words.
 - `stripnum`: indica se considerare o meno i numeri.
 - `minsize`: la lunghezza minima delle parole da considerare.
 - `stemming`: indica se applicare lo stemming o no.
- **fit**: questo metodo non esegue nulla, poiché non è necessario creare un modello. Viene comunque implementato perché fa parte dell'API di `sklearn`.
- **transform**: prende come input i vettori (il training set). La prima operazione è creare una copia dei documenti originali. Sulla copia, si applica la funzione `map`, che applica una funzione a tutti gli elementi della lista. Questa funzione si occuperà della trasformazione del testo.
- **transformText**: questo metodo prende in input un documento di testo e, attraverso una serie di condizioni (`if`), applica o meno gli step di pulizia in base ai valori degli attributi ricevuti dal costruttore (come lo stemming o la rimozione delle stop words).

Non è necessario implementare `fit_transform`, poiché viene ereditato da `TransformerMixin`

Questo metodo combina automaticamente le operazioni di `fit` e `transform`, utilizzando le versioni ridefinite dei metodi `fit` e `transform` nella nostra classe

Nell'**esempio** del professore, dopo aver creato la pipeline, possiamo passare direttamente gli iperparametri necessari (se non vogliamo eseguire la calibrazione), e successivamente applicare solo `fit` e `predict`.

È come scrivere solo due funzioni: una per l'addestramento e una per la predizione.

PRINCIPI DEL SENTIMENT E EMOTION MINING

Il **Sentiment Mining** e l'**Emotion Mining** sono tecniche utilizzate per analizzare e comprendere le emozioni e i sentimenti espressi in un testo.

L'obiettivo principale è determinare se un testo (ad esempio, un post sui social media o una recensione) esprime sentimenti positivi, negativi o neutri e identificare eventuali

emozioni come felicità, rabbia, tristezza, ecc.

A cosa serve analizzare un media per fare sentiment ed emotion analysis?

Analizzare un media attraverso il sentiment ed emotion mining è utile per:

- capire l'opinione pubblica su un determinato argomento o prodotto.
- monitorare il feedback degli utenti per valutare la percezione di un brand.
- individuare trend emotivi associati a eventi specifici o a discussioni online.
- identificare stati emotivi associati ai messaggi, come rabbia, paura o felicità, per analizzare l'impatto psicologico di un testo.

Come si applica la Sentiment Analysis?

La **Sentiment Analysis** si applica tramite l'analisi di parole, frasi o interi documenti, valutando il contenuto dal punto di vista emozionale e sentimentale.

Ogni parola ha molteplici attributi:

- il **sense** (significato) della parola, ossia il suo senso letterale.
- la **connotazione**, che indica se la parola ha un'accezione positiva o negativa dal punto di vista del sentiment e se rappresenta un'emozione specifica.

Connotazione ed emozione

Il concetto di **emotion** è complesso. In letteratura sono state definite sei emozioni di base:

- sorpresa
- felicità
- rabbia
- paura
- disgusto
- tristezza

Queste emozioni possono essere rappresentate su una scala, formando una sorta di stella con otto punte, dove emozioni opposte si trovano agli antipodi. Ad esempio:

- **noia** è opposta a **Preoccupazione**.
- **interesse** è opposto a **Distrazione**.

Rappresentazione delle emozioni su un piano cartesiano

Le emozioni possono essere rappresentate anche su un piano cartesiano:

- **asse X (Valenza)**: indica se l'emozione ha una connotazione negativa o positiva. Una valenza negativa implica emozioni con una connotazione negativa (ad esempio, rabbia o tristezza).
- **asse Y (Arousal)**: rappresenta l'intensità dell'emozione (ad esempio, una forte rabbia avrà un arousal alto, mentre una leggera tristezza avrà un arousal basso).

Come catturare un'emozione?

Il modo più semplice per catturare un'emozione è fare un'**analisi per keyword**, ovvero identificare se un testo contiene parole chiave che esprimono una specifica emozione o sentimento.

In letteratura sono stati sviluppati dizionari che associano ad alcune parole un **sentiment** (positivo o negativo) o un'emozione specifica. Tuttavia è il **contesto** che influisce molto sul sentimento e sull'emozione espressi da una parola

Come funziona un classificatore di emozioni?

Un classificatore di emozioni funziona assegnando a ogni parola in un testo diversi livelli di emozione (tra le otto emozioni di base) e un sentiment positivo o negativo.

Ogni parola può essere associata a più emozioni contemporaneamente e, in base all'analisi complessiva, si può determinare se il sentimento generale del testo è positivo o negativo.

Esempio di Emotion Mining

Prima di utilizzare approcci di **machine learning** per il sentiment e emotion mining, possiamo sperimentare con librerie già esistenti, come `text2emotion`.

Questa libreria effettua un'analisi delle emozioni di un testo

È consigliabile installare la versione 1.6.3, poiché alcune funzioni sono state deprecate nella versione più recente.

Un approccio utile nel preprocessing è quello di fare **lemmatization** e utilizzare espressioni regolari per riconoscere emoji create con segni di interpunkzione, sostituendole con la codifica Unicode corrispondente. In alternativa, per essere sicuri, si potrebbe sostituire l'emoji con una parola chiave (ad **esempio**, sostituire un'emoji di felicità con la parola "happy").

SENTIMENT ANALYSIS (Analisi del Sentimento)

L'analisi del sentimento consiste nel classificare un testo in base a un'emozione: positivo, negativo o neutro.

Questi sentimenti possono avere diverse intensità. Spesso si utilizza una classificazione binaria (vero/falso), oppure una scala a 3 valori che include il neutro, o ancora scale con più valori (ad **esempio**, 5 valori)

Classificare su scale più complesse è difficile: mentre è facile dire se un testo è positivo o negativo, è più complicato decidere se un sentimento è, ad esempio, un "-2" o un "-1".

Approcci all'analisi del sentimento:

- **Dictionary-based:** usa dizionari di parole predefinite con sentimenti associati.
- **Natural Language Parsing:** analizza l'intera frase per comprenderne il contesto, non solo le singole parole.
- **Machine Learning (ML):** approcci basati su algoritmi di apprendimento automatico per classificare il sentimento.

Un **esempio** di strumento disponibile è l'**NLTK** `SentimentIntensityAnalyzer`, che può riconoscere anche le emoji fatte con segni di interpunkzione.

Considerazioni per il preprocessing

- Non è sempre ideale rimuovere tutta la punteggiatura.

- Sostituire emoji con parole chiave.
- Rimpiazzare segni di interpunkzione particolari:
 - "!!" → `BigExclamation`
 - "!" → `Exclamation`
 - "?!?" → `Doubtful`

Uppercase

Se una parola è in maiuscolo, potremmo farla precedere da "shout" prima di trasformarla in minuscolo. In Python, possiamo usare la funzione `isupper()` per rilevare le parole in maiuscolo.

Parsing del testo HTML

La libreria `BeautifulSoup` è utile per analizzare testo HTML, con il metodo `get_text()` che permette di estrarre il testo eliminando i tag HTML

WORD SEMANTICS (Semantica delle Parole)

Fino ad ora abbiamo trattato le parole come unità indipendenti

Modelli come gli **n-grammi** e l'**LSI** considerano un minimo di semantica

Ogni parola è composta da:

- **Lemma:** Una sequenza di caratteri (una parola), con associata la parte del discorso (nome, verbo, ecc.).
- **Sense:** Il significato della parola, che può essere multiplo.

Concetti importanti:

1. **Polisemia:** Un lemma che ha più significati.
2. **Sinonimia:** Più parole che hanno lo stesso significato.
3. **Similarità:** Parole che appartengono allo stesso dominio semantico (es. "car" e "bike").
4. **Relazioni tra parole:**
 - **Antonomia:** Parole opposte.

- **Iponimia:** Una parola il cui significato include quello di un'altra (es. "mouse" può essere un animale o un dispositivo).
- **Meronimia:** Una parola che descrive una parte di un'altra (es. "becco" è una parte di un "uccello").
- **Olonimia:** Descrive un insieme o tutto (es. "casa" è l'insieme di "porta", "finestra", ecc.)

Nota: Applicare la transitività a queste relazioni non è sempre intuitivo (ad esempio, dire che una "casa ha una maniglia" può sembrare strano).

WORD RELATIONSHIP ANALYSIS CON WORDNET

WordNet è un database lessicale usato per operazioni di lemmatizzazione.

Crea una rete di grafi e relazioni etichettate tra le parole.

Alcune caratteristiche di WordNet:

- rappresenta le relazioni tra le parole attraverso archi.
- crea una **matrice lessicale**: le righe rappresentano i significati, le colonne i lemmi. All'intersezione di righe e colonne ci sono gli elementi di WordNet.
- se più lemmi sono sulla stessa riga, sono sinonimi. Se lo stesso lemma appare su più righe, ha più significati (es. "mouse").

Come rappresentare i concetti in WordNet

Potremmo usare sinonimi, ma WordNet utilizza un approccio basato su **synsets** (insiemi di sinonimi). Ogni concetto è rappresentato da una definizione simile a quella del dizionario, accompagnata da una lista di parole associate al concetto

▼ Lezione 17 - 06/05/2024

Un **synset** è un insieme di tutti i significati che una parola può avere, accompagnati dai sinonimi per ogni significato

Per esempio, se cerchiamo la parola "travel", possiamo ottenere:

- la **part-of-speech** (parte del discorso, ad esempio nome, verbo, ecc.)
- il **meaning** (significato) associato alla parola

- un **esempio di frase** che utilizza quella parola in quel contesto

Accesso a relazioni lessicali

- **Iperonimi:** Termini più generici che comprendono quello in esame (ad esempio, "veicolo" per "auto").
- **Iponimi:** Termini più specifici (ad esempio, "berlina" per "auto").

Quando otteniamo un synset, possiamo trovare più **lemmi** associati, che rappresentano i sinonimi per quel significato. Per ogni lemma, possiamo inoltre accedere alla lista dei **contrari** (antonimi)

È possibile che all'interno della lista dei lemmi ci siano dei duplicati, poiché un lemma può avere significati leggermente diversi in base al contesto.

WordNet offre un'API per calcolare la **similarità semantica** tra due synset.

Questo ci permette di quantificare quanto due concetti siano semanticamente vicini, usando diverse metriche

Esempio:

```
w1 = wordnet.synset('ship.n.01')
w2 = wordnet.synset('boat.n.01')
w1.wup_similarity(w2)
```

Un problema con WordNet è che non è aggiornato per includere termini moderni o brand comuni

Inoltre, **WordNet** non è un **database lessicale di dominio specifico**, ma piuttosto uno strumento **generico**, quindi può non essere sufficientemente dettagliato o preciso per applicazioni in domini specializzati (ad esempio, medicina, tecnologia, ecc.)

WORD EMBEDDINGS

L'obiettivo del **word embedding** è di attribuire una **semantica** alle parole basandosi sui dati, cioè imparare il significato di una parola dai documenti piuttosto che prendere

questa informazione da un dizionario costruito manualmente

Concetti fondamentali del word embedding

- **Distribuzione e contesto:** il significato di una parola è determinato dalla sua distribuzione nei documenti, in particolare dalle parole che appaiono vicino ad essa (contesto).
Questo concetto è simile a quanto fa l'**LSI** (Latent Semantic Indexing).
- **Rappresentazione spaziale:** le parole non vengono più trattate come singole dimensioni in un vettore, dove ogni parola sarebbe indipendente e ortogonale.

Ora vogliamo che ogni parola sia rappresentata come un punto in uno **spazio multidimensionale**.

Rappresentare le parole in uno spazio multidimensionale

Invece di trattare le parole come singoli elementi indipendenti, le rappresentiamo come punti in uno spazio con molte dimensioni (**centinaia di dimensioni**)

In questo spazio:

- **ogni parola è rappresentata da un vettore.**
- le parole che appaiono in contesti simili saranno vicine nello spazio vettoriale, creando una sorta di **clustering semantico**.

Questa è una rappresentazione standard utilizzata nei modelli di **Natural Language Processing (NLP)**.

Discutiamo ora di due modelli fondamentali per la rappresentazione vettoriale delle parole:

- **TF-IDF (Term Frequency-Inverse Document Frequency):**
 - TF-IDF assegna a ogni parola un peso che rappresenta quanto sia importante una parola in un documento rispetto all'intero set di documenti.

- In questa rappresentazione, i vettori sono **sparsi**, cioè hanno molti coefficienti nulli

- **Word2Vec:**

- In **Word2Vec**, le parole sono rappresentate da vettori **densi** (cioè, con coefficienti non nulli e valori numerici significativi).
- Ogni documento è rappresentato come un insieme di vettori, uno per ogni parola.
- I coefficienti dei vettori sono i parametri di un **modello predittivo** che cerca di predire una parola basata sulle parole che la circondano nel testo (il contesto).

Concetti chiave del word embedding

1. **Il significato di una parola è determinato dalle parole che compaiono vicine ad essa** (contesto).
2. **Ogni parola è un punto in uno spazio multidimensionale**, dove vicinanza indica similarità semantica.

In questo spazio, parole con significati simili tendono ad apparire vicine, formando dei **cluster**. Ad **esempio**, parole come "king" e "queen" avranno vettori vicini, perché appaiono spesso in contesti simili.

TERM-DOCUMENT MATRIX

Inizialmente, possiamo rappresentare un documento come un **vettore di parole** (bag of words).

Tuttavia, con il **word embedding**, vogliamo rappresentare le parole come **vettori** basati sul modo in cui queste occorrono nei documenti.

L'idea di base nel word embedding è che due parole sono **simili** se i loro **vettori di contesto** sono simili.

In altre parole, due parole come "tea" e "coffee" sono semantiche vicine perché appaiono in contesti simili.

Perché utilizzare vettori densi?

Utilizzare **vettori densi** (cioè vettori con valori non nulli) **permette di calcolare più facilmente la similarità tra due parole**

Il modello **Word2Vec** non si basa sul contare quante volte una parola appare, ma costruisce un **classificatore** che cerca di predire una parola basandosi su quelle vicine (contesto). Questo approccio utilizza la tecnica **skip-gram**, in cui una parola viene presa e il modello cerca di predire le parole vicine, cioè quelle che appaiono nel suo contesto

La **differenza** principale tra **Word2Vec** e modelli basati su **SVD (Singular Value Decomposition)** è che Word2Vec impara da **contesti ridotti** (finestre di parole vicine), mentre l'SVD analizza l'intero documento

Una caratteristica fondamentale di Word2Vec è il concetto di **self-supervision**: non abbiamo bisogno di un dataset etichettato in modo esplicito, poiché il modello si auto-supervisiona.

Il predittore generato da questo modello avrà dei **coefficienti** che costituiscono la **rappresentazione vettoriale** della parola.

Nel modello **skip-gram**, prendiamo una **parola target** e consideriamo una **finestra di contesto** (ad **esempio**, di dimensione 2).

In questo caso, le parole vicine alla parola target (due precedenti e due successive) sono utilizzate per costruire esempi positivi

Esempio per la parola "albicocca":

- Input: le due parole precedenti e le due parole successive.
- Obiettivo: predire la parola target basandosi sul contesto.

Questo modello è probabilistico (non bayesiano) e si basa su **predizione logistica**: stimiamo la probabilità che una parola compaia in una certa sequenza.

Per ogni parola, viene costruito un modello predittivo

Tuttavia, non dobbiamo farlo manualmente, perché possiamo usare **modelli di embedding pre-addestrati** che esistono già.

L'obiettivo del modello è predire che una parola compaia in una certa posizione nel contesto e **minimizzare la similarità** con parole non correlate (parole negative).

In altre parole, il modello cerca di avvicinare i vettori delle parole simili e allontanare quelli delle parole non correlate.

Riassunto del processo

- **Corpus di documenti:** Si parte da un insieme di documenti.
- **Gruppo di parole:** Si prendono gruppi di parole attorno a una parola target.
- **Classificatore:** Si addestra un classificatore per predire la probabilità che una parola appaia vicino a quella target.
- **Coefficiente vettoriale:** Una volta addestrato il classificatore, estraiamo i **coefficienti** dai modelli per ottenere la rappresentazione vettoriale delle parole.
- **Ripetizione per ogni parola:** Questo processo viene ripetuto per ogni parola nel dataset.

Alla fine, otteniamo un vettore per ogni parola, dove parole simili avranno vettori simili.

Ad **esempio**, calcolando la similarità del coseno tra i vettori di "albicocca" e "cilegia", otterremo un valore alto, indicando che i due concetti sono semantici vicini.

Durante l'addestramento di un modello di embedding, è possibile specificare la **dimensione della finestra di contesto**, cioè il numero di parole da considerare attorno alla parola target per fare le predizioni.

RELAZIONI ANALOGICHE

Le **relazioni analogiche** sono analogie tra concetti rappresentati da vettori nel word embedding.

Consideriamo due parole: "**mela**" e "**albero**". Queste due parole sono rappresentate come vettori nello spazio di embedding.

Se tracciamo un vettore che va da "mela" a "albero" e ci poniamo la domanda: "se 'mela' sta a 'albero', 'uva' sta a cosa?", possiamo cercare un vettore parallelo che ci porti a "**vite**"

Problemi di bias nei modelli

Questi modelli possono rivelare pregiudizi

Ad **esempio**, se prendiamo l'analogia `papà:medico = mamma:x`, il modello **preaddestrato** potrebbe restituire **infermiera**, riflettendo un pregiudizio di genere

Tutti i modelli preaddestrati soffrono di questi **bias**.

Strumenti come **ChatGPT** sono addestrati per ridurre al minimo questi pregiudizi, cercando di rispondere in modo neutrale.

PROBLEMS

Se volessimo utilizzare i word embedding con i modelli tradizionali (come per esempio per fare sentiment analysis con **Naive Bayes**), avremo alcune difficoltà

I modelli che abbiamo visto finora trattavano i documenti come matrici di termini e documenti, dove ogni parola era una colonna e ogni documento era una riga.

Con i word embedding, le parole sono rappresentate da **vettori** e **NON** da **singole etichette**

Per gestire i word embedding nei modelli, possiamo usare un **embedding tensor**

Invece di rappresentare ogni parola con un'etichetta, passiamo al modello un **vettore per ogni parola**

Dovremmo quindi passare **n vettori** (uno per ogni parola nel documento).

Per rappresentare un documento, non utilizziamo più una **matrice termine-documento** tradizionale, ma rappresentiamo ogni parola nell'ordine con cui appare, utilizzando il suo embedding

Non c'è bisogno di effettuare lo stemming, ma si può decidere se rimuovere o meno le stop word

Pertanto, ogni documento è una matrice di dimensioni $m \times n$, dove:

- m è la dimensione dell'embedding;
- n è la lunghezza del documento.

Poiché i documenti possono avere lunghezze variabili, consideriamo n come la lunghezza massima.

Problema di avere documenti di lunghezza variabile

Un problema sorge dal fatto che i documenti hanno **lunghezze diverse**, quindi ogni documento avrà un numero differente di parole (e di conseguenza di vettori)

Per affrontare questo problema potremmo:

- **troncare**: se un documento supera una certa lunghezza massima, lo si tronca.
- **padding**: se un documento è più corto, si aggiungono **vettori di padding** (vettori nulli) per raggiungere la lunghezza richiesta

Questa tecnica viene utilizzata anche nei modelli di **Large Language Models (LLM)** per gestire input di lunghezza variabile

Il **document embedding** è una rappresentazione dell'intero documento

Invece di lavorare con i vettori delle singole parole, si può creare un embedding del documento facendo la **media dei vettori** delle parole che lo compongono.

Questo permette di rappresentare l'intero documento come un unico vettore, utilizzabile in vari modelli di machine learning.

Se volessimo usare i word embedding con **Naive Bayes**, avremmo il seguente problema: Naive Bayes non funziona bene con i vettori che contengono **coefficienti negativi**, poiché il modello si basa su **probabilità** e i coefficienti negativi non sono interpretabili in quel contesto.

Per usare Naive Bayes, quindi, è necessario **riscalare i vettori** in modo da **rimuovere i valori negativi**

Al contrario, **SVM** (Support Vector Machines) **può lavorare direttamente con i vettori densi di word embedding senza problemi**, perché non richiede che i dati siano in forma probabilistica

▼ Lezione 18 - 09/05/2024

INTRODUCTION TO NEURAL NETWORK

Il concetto di funzionamento di una **rete neurale** è quello del **neurone**, o meglio del cervello umano che è composto da miliardi di neuroni

Le prime reti neurali erano costituite da decine di neuroni

Oggi con i LLM siamo arrivati ad avere modelli con centinaia di miliardi di connessioni

Il neurone di un essere vivente è dotato di un nucleo che riceve impulsi da altri neuroni connessi o dal sistema nervoso

In generale, un **neurone riceve input attraverso** un canale, detto **dendrite**

I **segnali** che riceve il neurone sono **impulsi elettrochimici**

In **output**, un neurone produce un segnale e questo viene trasmesso ad altri neuroni tramite un canale, detto **assone**

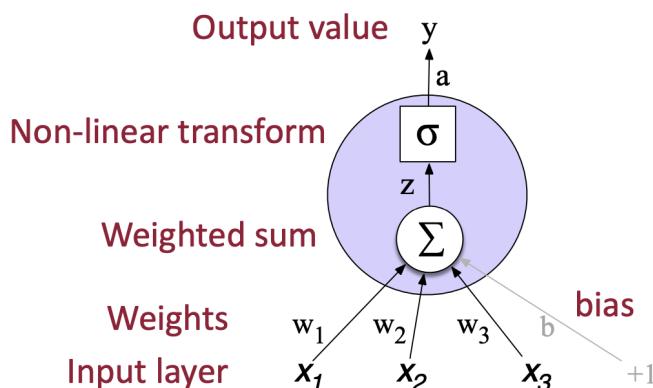
La **parte finale dell'assone si divide** in tanti rami che prendono il nome di **sinapsi**

L'idea dell'IA è **unire** i vari **segnali d'ingresso** e produrre un'uscita attraverso un **procedimento matematico**

In una rete **neurale**, un neurone ha degli **ingressi** (i dentriti), i quali hanno una certa importanza (**un peso**)

Oltre agli ingressi veri e propri, esiste un **ulteriore termine**, detto **bias**

Il concetto di bias qui è completamente diverso dal pregiudizio dei modelli predittivi, infatti in questo caso il **bias** è il **termine noto dell'equazione**, una **costante**



Il **nucleo** del neurone artificiale **effettua** una **somma pesata** degli **ingressi**

$$z = b + \sum_i w_i x_i$$

L'output del nucleo quindi è proprio la somma pesata, ma **NON** è ciò che viene **invia**to al **neurone successivo**

Infatti, prima viene effettuata una **trasformazione**, spesso **NON lineare**, indicata con σ

Le decisioni da prendere per costruire una rete neurale sono:

- **topologia della rete**, cioè da **quanti neuroni** è formata la rete, **come** sono **connessi**, **quanti strati** di neuroni, ecc
- **funzione di attivazione**, bisogna decidere la funzione σ , **NON** è detto che si utilizzerà la **stessa funzione** per **tutti i neuroni** della rete
- **algoritmo di training**

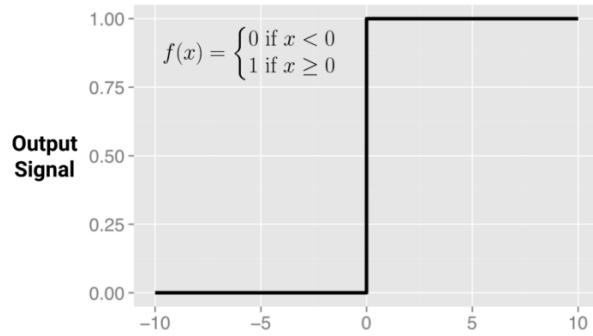
FUNZIONE DI ATTIVAZIONE

Questa funzione **deve permettere** al **neurone** di **prendere la decisione**

La **decisione** più semplice che un neurone potrebbe prendere è quella di fare una **classificazione**, cioè il neurone risponde **vero** o **falso** (o meglio 0 o 1 dato che lavora con i numeri)

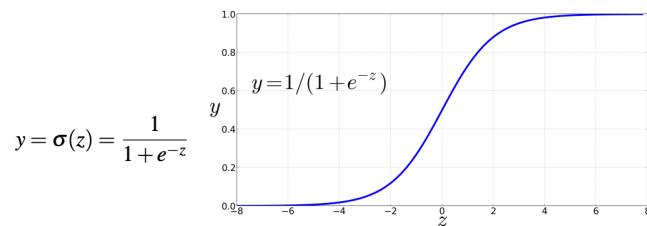
FUNZIONE A GRADINO

La funzione più elementare che si potrebbe scegliere è una **funzione a soglia**, cioè **se l'input è maggiore** di una **soglia**, allora il neurone **risponde** con **1**



In realtà, in letteratura si è provato a definire una funzione di attivazione che calcola meglio la relazione input-output

SIGMOIDE LOGISTICA



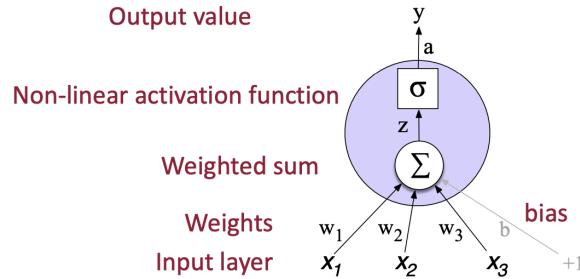
È una **funzione esponenziale**, quindi è **NON lineare**

Il valore all'**inizio** è **basso**, mentre quando arriva **intorno** alla **soglia** inizia a **salire fino a saturare**

Se avessimo una rete neurale costituita da un solo neurone, la rete compie quest'operazione sui dati in ingresso:

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

Esempio: supponiamo di avere una rete neurale a singolo neurone



e di avere i seguenti valori dei pesi:

$$w = [0.2, 0.3, 0.9] \\ b = 0.5$$

Proviamo a dare in ingresso il seguente vettore x :

$$x = [0.5, 0.6, 0.1]$$

L'uscita sarà:

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{1}{1 + e^{-0.5 \cdot 2 + 0.6 \cdot 3 + 0.1 \cdot 9 + 0.5}} = \frac{1}{1 + e^{-0.87}} = 0.7$$

La rete produce 0.7, quindi andrà a mettere la **soglia** su questo valore (un po' come fa il classificatore bayesiano)

b viene **calibrato esattamente** come i **pesi**, quindi sarà un peso a tutti gli effetti

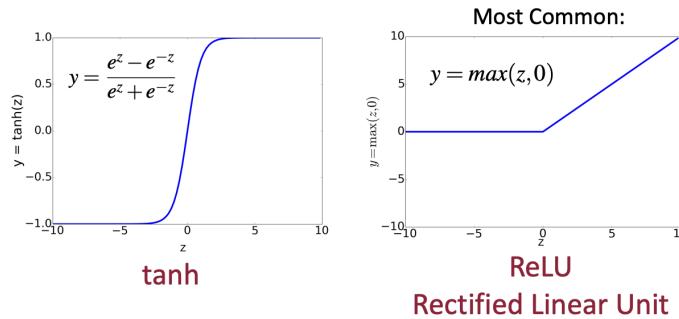
Il **singolo neurone** è proprio un **classificatore a regressione logistica**

Altre funzioni di attivazione:

- **lineare**
- **saturated linear**
- **tangente iperbolica** (molto simile alla regressione logistica)
- **gaussiana** (utilizzata raramente)
- **ReLU (Rectified Linear Unit)**

Le **funzioni di attivazione** più utilizzate sono:

- **tangente iperbolica** (sale più rapidamente della regressione logistica)
- **funzione ReLU** (è **costante a 0** fino alla **soglia** e poi **cresce linearmente**)



La funzione **ReLU** è molto utilizzata perché il **funzionamento** di una **rete neurale** consiste nel **calcolare derivate** e per questo tipo di funzione è molto semplice fare il calcolo

Le prime reti neurali utilizzavano solo modelli logistici, mentre le reti neurali di **oggi** usano **funzioni logistiche solo** per i **neuroni di uscita**

Invece, i **neuroni nascosti** utilizzano **ReLU** perché in questo modo si rende l'**addestramento** molto **più efficiente**

Un altro aspetto da considerare è che queste **funzioni di attivazione NON** sono **bounded (limitate)**, quindi a seconda degli ingressi potremmo avere problemi ad interpretare le uscite, soprattutto se ci sono molteplici funzioni di attivazione

Allora le **reti neurali** vanno a **normalizzare gli ingressi** in un certo **range**, in questo modo è possibile anche confrontare gli **ingressi**

THE XOR PROBLEM

La prima domanda che dobbiamo porci quando vogliamo costruire una rete neurale è "mi basta un singolo neurone?" oppure "mi basta un solo livello di neuroni?"

Il problema dello **XOR** è un problema alla base delle reti neurali più complesse

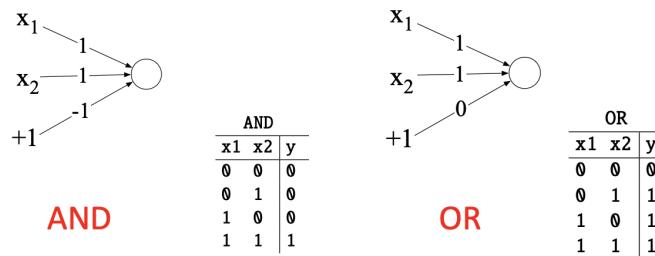
L'implementazione degli operatori booleani è la cosa più semplice da voler far fare ad un calcolatore

Proviamo a fare operazioni di **AND**, **OR** e **XOR** ad una rete neurale e utilizziamo una **funzione a gradino**

$$y = \begin{cases} 0 & \text{se } w \cdot x + b < 0 \\ 1 & \text{se } w \cdot x + b \geq 0 \end{cases}$$

Cioè l'uscita sarà 0 se l'uscita del neurone sarà minore della soglia e 1 viceversa

Sia per realizzare **AND** sia per **OR** è necessario un **singolo neurone**



In particolare, per l'**AND pesiamo 1 gli ingressi**, mentre il **terminie noto** lo pesiamo **-1**

In questo modo **se uno** dei due vale **0**, la somma vale 0, per cui la funzione di attivazione resta **0**

Invece se entrambi gli ingressi sono **1**, la somma vale 1 e la funzione di attivazione sarà **1**

Per l'**OR pesiamo il termine noto 0**

Invece lo **XOR NON** si può realizzare con un **singolo neurone**, il perché si spiega in maniera geometrica

Supponiamo di avere due ingressi x_1 e x_2 e poniamo l'equazione di output del nucleo a 0 per ricavare x_2 in funzione di x_1

$$w_1x_1 + w_2x_2 + b = 0$$

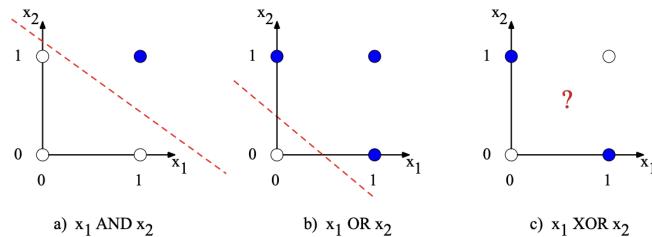
$$x_2 = \left(\frac{-w_1}{w_2}\right)x_1 + \left(\frac{-b}{w_2}\right)$$

Otteniamo la funzione di una **retta**

I possibili valori di x_2 sono 4 punti

Nel caso dell'**AND** e dell'**OR**, la retta separa **perfettamente** i punti

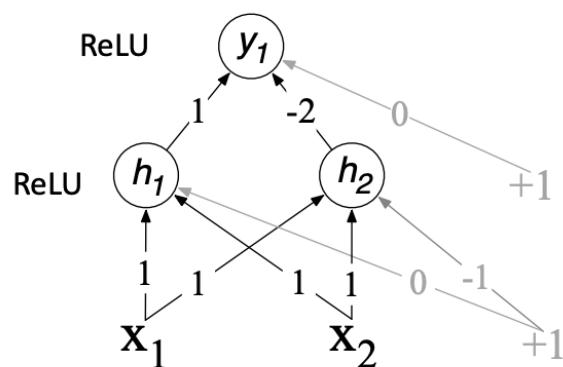
Mentre per lo **XOR** **NON** esiste una retta del genere, infatti si dice che lo **XOR** è una funzione **NON linearmente separabile**



In pratica, questo è lo stesso problema che abbiamo con SVM, quindi la rete neurale fa ciò che fanno i kernel nell'SVM

Per risolvere il problema, possiamo creare nuove variabili che sono funzioni delle precedenti **aggiungendo dei nuovi nodi**

XOR		
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



In particolare, **aggiungiamo due nodi** che vanno a connettersi ad uno **successivo**

La **rete** neurale sarà a **due livelli**, dove il **primo** sarà **nascosto** e **connesso all'input**

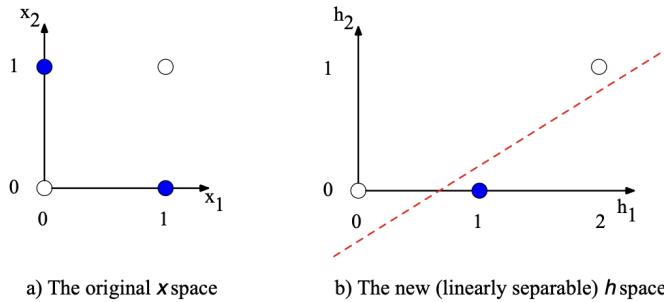
Il **primo neurone** sarà h_1 e avrà in ingresso x_1 e x_2 con **peso 1**, mentre il **termine noto** sarà **0**

Invece h_2 avrà **pesi 1** per x_1 e x_2 ma **termine noto -1**

In pratica, abbiamo un nodo che effettua un **OR** e un altro che effettua un **AND**

h_1 e h_2 li connettiamo con **pesi** rispettivamente **1** e **-2** ad un nuovo nodo con **termine noto 0**

In pratica, abbiamo fatto un **OR tra una variabile e il suo negato**



Il piano non è più x_1-x_2 ma sarà h_1-h_2

Così come con gli SVM aggiungevamo variabili con il kernel, ora **aggiungiamo layer** (cioè facciamo prendere più decisioni) e passiamo ad uno spazio h_1-h_2 che è **linearmente separabile**

Una rete **feed forward** è una rete in cui gli **ingressi transitano** nei neuroni **verso l'uscita senza feedback**

Ma tutte le reti sono feed forward?

Innanzitutto, dobbiamo capire quanti input ha la rete

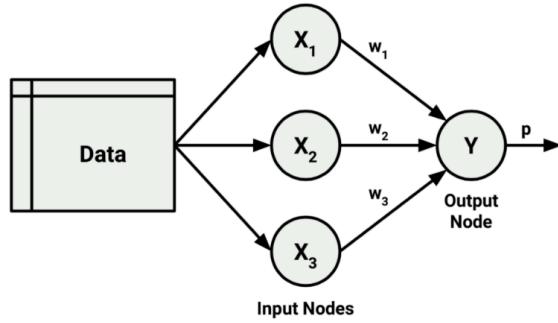
Dato che nel caso di questo corso trattiamo testo, il numero di input è variabile

Quindi **stabiliamo a priori la dimensione del testo** che la rete neurale prende in input (stabiliamo una lunghezza massima)

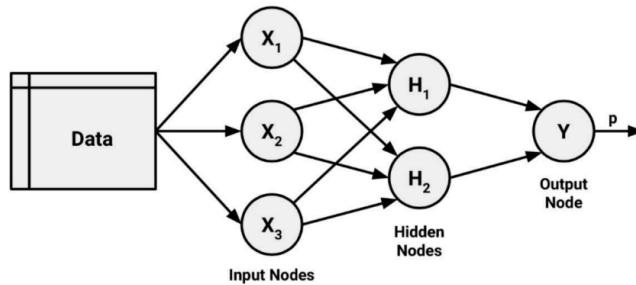
Se la **dimensione del testo** è più **piccola**, si **riempie** con tutti **0**

Invece, se la **dimensione del testo** è più grande bisogna necessariamente a **tagliare**

SINGLE LAYER NETWORK



MULTIPLE LAYERS



3 ingressi, 2 nodi nascosti e un unico nodo di output

DIREZIONE DEI DATI

Nelle reti viste fin ora, i dati vanno sempre **dagli ingressi alle uscite**

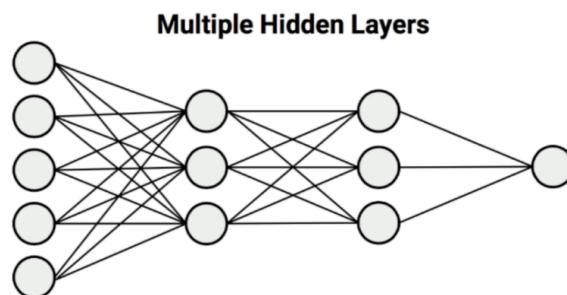
A cosa può servire far viaggiare i dati anche **all'indietro**?

Così come nei latch e flip-flop, la **retroazione** serve per dare **memoria**, per **ricordare lo stato**

Ciò è utile perché vogliamo che una parola venga interpretata sulla base delle parole processate precedentemente, quindi la rete deve avere memoria di ciò che ha visto precedentemente

DEEP NEURAL NETWORK (DNN)

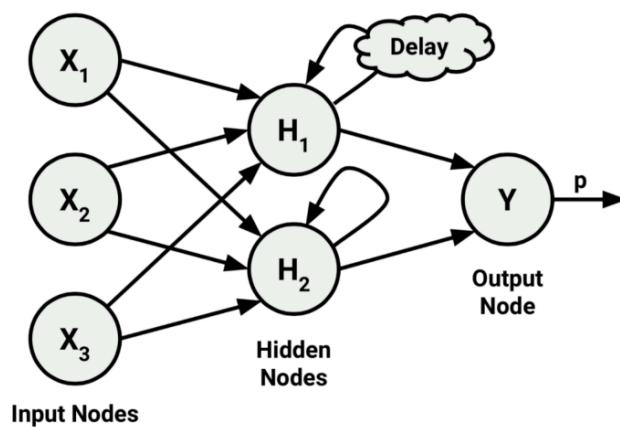
Tipicamente una rete che ha **più layer nascosti** (da 2 in poi) è detta **rete neurale profonda**



RECURRENT NETWORK

È una rete che permette all'**output** di un neurone di **tornare indietro**, cioè di essere dato in **input** al **neurone stesso** o ad **altri** neuroni precedenti

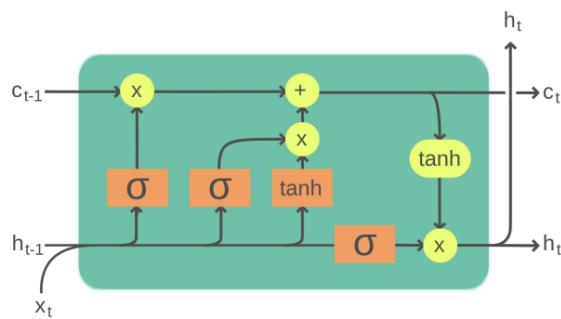
In questo modo si possono **processare sequenze** di eventi



Un esempio di blocco di rete neurale profonda è il cosiddetto **LSTM**

LSTM (Long Short Term Memory) NETWORK

Questo tipo di blocco è abbastanza famoso perché è il blocco **più piccolo** presente in una **rete ricorrente** e quindi delle reti **transformer** che sono alla base dei **LLM**



c = stato

h = uscita

La rete prende in **ingresso l'input**, lo **stato** e l'**uscita precedente** e produce in **output** sia lo **stato** sia l'**uscita successiva**

Tipicamente, in una rete che processa testo ci saranno tanti blocchi LSTM, in modo da poter **processare una parola in base alle precedenti**

I blocchi **successivi tengono conto** di tutti gli **stati precedenti** perché l'output di uno dipende da quello precedente, ma questo a sua volta dipende da quello precedente ancora e così via...

Ovviamente **più si avanza, più l'effetto** degli **stati iniziali va a scemare**

MULTILAYER PERCEPTION (MLP)

È la rete neurale **unidirezionale (feed forward)**, cioè gli input vanno solo in una direzione

I **nodi** di **input** vanno **settati** in base al **numero** di **input** che ci sono

Invece, il **numero** di **nodi** di **output** dipende da cosa andiamo a costruire

La **dimensione** dei **layer interni** invece va determinata **sperimentalmente**

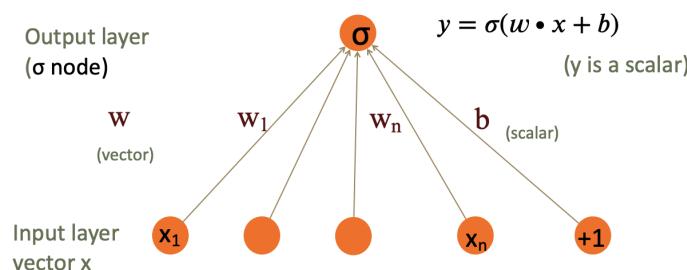
FEEDFORWARD NEURAL NETWORK

Questo tipo di rete presenta **uno strato nascosto**

Il caso più semplice di una rete di questo tipo è la **regressione logistica**

L'**output** è un valore compreso tra **0 e 1**

Viene utilizzata la **funzione logistica** perché **trasforma l'output** di un nodo in **qualcosa che assomiglia ad una probabilità**



Che succede quando devo fare una **classificazione a più livelli**?

NON basta più un **unico nodo**

Esempio: quando si fa sentiment analysis si possono avere documenti o positivi o negativi oppure neutri

Si utilizza la regressione **SOFTMAX**

Un **classificatore a 3 livelli** non produce una sola probabilità, ma **3 valori**, cioè:

- **probabilità** che il documento sia **positivo**
- **probabilità** che il documento sia **negativo**
- **probabilità** che il documento sia **neutro**

La condizione è che la **somma** delle **probabilità** deve essere **1**

$$P(\text{positive}|\text{doc}) + P(\text{negative}|\text{doc}) + P(\text{neutral}|\text{doc}) = 1$$

In generale, ci saranno **tante uscite quante** sono le **categorie** e ogni **uscita** sarà il **risultato** di una **funzione logistica**

La **rappresentazione softmax** è **diversa** da una **rappresentazione binaria canonica** perché ora utilizziamo **3 uscite** e non 2 (perché ci servirebbero 2 bit per una rappresentazione binaria)

È come se avessimo qualcosa di **analogico**, quindi è più affidabile

Ogni layer di uscita della trasformazione sarà formata da un softmax che avrà tante uscite quant'è la dimensione del vocabolario

Si va a prendere il **massimo** del **vettore** di uscita perché significa quella è la classe più probabile a cui appartiene il vettore z

LOGISTIC REGRESSION IN PYTHON

Esiste una classe implementata in `sklearn.linear_model`

Si utilizza esattamente come gli altri classificatori

TORNIAMO ALLE RETI NEURALI

Adesso andiamo fare una piccola modifica, cioè andiamo a considerare il **termine noto** come un **peso** esattamente come gli altri, con la differenza che non è collegato ad un input vero e proprio ma ad una costante

Questa cosa è solo un aspetto di notazione, ma in generale ci serve perché dobbiamo stimare i valori dei pesi perché sono quelli che fanno **funzionare** la rete

Però i pesi **NON** si conoscono, ma si **ottengono** con l'**addestramento** della rete

e qui vengono i dolori sia dal punto di vista concettuale che computazionale

Una rete neurale una volta addestrata è banale, perché fa somme pesate, il problema è proprio addestrarla

FASE DI INFERENZA: si usa una rete già addestrata per fare una classificazione

COME FUNZIONA IL TRAINING

La fase di training si divide in **due step**:

- **forward**
- **backward**

Inizialmente si parte con dei **pesi random** (in realtà non sempre)

Ci sono dei **dati etichettati** che vengono dati in **ingresso** e si **produce la classificazione in uscita**

A questo punto, si **confrontano le predizioni** con i **valori attesi**

La **distanza** tra la **predizione** e i **valori attesi** crea la **funzione di loss** (di perdita)

In base a questa funzione, si capisce se bisogna migliorare la rete, cioè modificare i pesi (ogni arco, connessione tra neuroni, è un peso)

COME MODIFICO I PESI?

I pesi vanno modificati considerando la funzione di loss

La **funzione di loss** è la **funzione complessiva** della rete, è un aggregato di funzioni perché ogni nodo diventa funzione del nodo precedente, quindi la **rete neurale** può essere vista come una **funzione matematica composta** ("di brutto")

Di conseguenza, la **funzione di loss** è una **funzione** della **rete stessa**

L'obiettivo è **minimizzare** la **funzione di loss**

Quindi in sostanza il tutto si riduce a minimizzare una funzione, il problema è che ci sono centinaia di migliaia di variabili

Per minimizzare bisogna calcolare le **derivate**, in particolare le **derivate parziali**

Per questo motivo si usa la **ReLU** come **funzione di attivazione**

COME AVVIENE LA PROCEDURA

Il training set viene diviso in tanti pacchettini

Queste **sub-parti** del training set prendono il nome di **batch**

Infatti, un **parametro** da definire per la rete neurale è il **batch size** che è dato dal numero di parametri di ingressi

In generale, se si creano **batch troppo grandi**, dato che questi vanno **caricati in memoria**, è possibile che non si possa lavorare perché non c'è memoria a sufficienza

Invece, con **batch piccoli** l'addestramento è più lungo perché ci saranno **più fasi di forward**

Dopo aver passato tutti i **batch** e aver **calibrato i pesi**, la rete ottenuta viene utilizzata con il **validation set**

Il **validation set** è fondamentale per le reti neurali

A questo punto, si calcolano le **metriche** (in particolare l'accuratezza) sul validation set

Se **NON** si è soddisfatti, si ripete **tutto** il processo da capo

Questo **processo** viene detto **epoca**

Per questo, va fissato anche il **numero di epoch**, cioè quante volte si fa **ripassare il training set** per **tutta** la rete

Però, dato che i dati sono sempre gli stessi, dopo un po' non succede più niente

Per evitare calcoli inutile esiste un meccanismo, detto **early stopping**, il quale **controlla** se dopo tot epoch **NON** si hanno **miglioramenti** significativi e nel caso **stoppa** il processo

Scendiamo più nel dettaglio

Passiamo dei dati di training alla rete e si fa un **passo di forward**, il quale produce l'output \hat{y} che viene **confrontato** con il **valore atteso**

Si calcola la **loss** tra il **valore atteso** e quello **stimato**

Il processo di **backward** calcola delle **derivate** per capire come **modificare** i pesi

Questo viene fatto attraverso un algoritmo di ottimizzazione che capisce come muoversi nello **spazio dei pesi (spazio n dimensionale)**

COM'È FATTA LA FUNZIONE DI PERDITA?

Si basa sul concetto di **entropia**

I decision tree che funzionano bene minimizzano l'entropia tra gli insiemi, cioè deve avere un gruppo di dati a bassa entropia, quindi una categoria prevale rispetto ad altre in un gruppo di dati

▼ Lezione 19 - 13/05/2024

La **loss function** in una rete neurale misura quanto bene il modello sta performando rispetto agli obiettivi attesi. L'obiettivo dell'addestramento è **minimizzare questa funzione**, così che il modello produca previsioni il più vicino possibile ai valori attesi.

Una delle loss function più utilizzate nelle classificazioni è la **cross entropy**

Entropia

L'**entropia** è una misura del disordine o dell'incertezza in un insieme di dati.

Nel contesto della classificazione, l'entropia misura quanto i dati siano distribuiti uniformemente tra le classi.

La formula dell'entropia è:

$$H(S) = \sum_{i=1}^c -p_i \log_2 p_i$$

Dove:

- S è un insieme di dati.

- c è il numero di categorie.
- p_i la probabilità che un dato appartenga alla categoria i .

Esempio:

Se le probabilità sono distribuite equamente (ad esempio, se abbiamo due eventi come "sole" e "pioggia", ciascuno con una probabilità del 50%), l'entropia è massima, e sarà pari a 1 (cioè, c'è massima incertezza).

Se invece, la probabilità di pioggia scende al 25%, l'entropia diventa più bassa: $H=0.8$, perché l'incertezza è diminuita.

La **cross entropy** si usa per **misurare la differenza tra due distribuzioni di probabilità** ed è utile quando **vogliamo confrontare** una distribuzione attesa (**la vera distribuzione dei dati**) con una distribuzione predetta (come quella **prodotta da un modello** di machine learning).

È il numero medio di bit necessari per identificare un evento quando si utilizza una codifica basata su una distribuzione q , anziché quella vera p

Immaginiamo di avere 8 eventi meteorologici con probabilità equiprobabili (ogni evento ha $p = 1/8$). In questo caso, servono **3 bit** per rappresentare ognuno degli 8 eventi.

Tuttavia, se alcuni eventi sono molto più probabili di altri (ad esempio, "sole" ha il 72% di probabilità e gli altri eventi hanno il 4%), possiamo usare meno bit mediamente per descrivere gli eventi.

In questo caso, l'entropia può scendere, e per esempio potremmo inviare **1.64 bit** in media, poiché è più efficiente trasmettere informazioni quando ci sono eventi molto probabili.

Cross Entropy in una Rete Neurale

Quando addestriamo una rete neurale, utilizziamo la **cross entropy** per **confrontare le previsioni del modello (q_i) con i valori attesi (p_i)**.

L'obiettivo è minimizzare la cross entropy, cioè far sì che le probabilità predette dal modello siano il più simile possibile a quelle attese.

Discesa del Gradiente

Per minimizzare la funzione di perdita, si utilizza un algoritmo di ottimizzazione chiamato **discesa del gradiente**

Discesa del Gradiente per un singolo nodo

Nel caso semplice di un singolo nodo (come nella **regressione logistica**), la funzione di perdita **ha un solo punto** di minimo globale.

Il processo di **addestramento si basa sul calcolo della derivata della funzione di perdita rispetto ai pesi del modello** per capire in quale direzione muoversi, cioè se aumentare o diminuire i pesi per ridurre l'errore

Il processo è il seguente:

1. si parte da un peso iniziale.
2. si calcola la derivata della funzione di perdita rispetto a quel peso.
3. si aggiorna il peso in direzione opposta alla derivata, per ridurre l'errore.

L'entità dello spostamento dipende da un parametro detto **learning rate**.

Il **learning rate** è un **iperparametro** che controlla quanto grandi sono i passi che facciamo nella discesa del gradiente.

Se il learning rate è troppo grande, rischiamo di **saltare** il minimo della funzione di perdita.

Per reti neurali con più dimensioni (più pesi), il gradiente è un **vettore** che contiene le derivate parziali della funzione di perdita rispetto a ciascun peso.

Gradiente per reti neurali

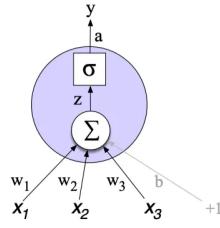
Se il modello ha n dimensioni, dobbiamo calcolare il **gradiente** della funzione di perdita **in ogni dimensione**, ossia le **derivate parziali rispetto a ciascun peso**

Questo ci consente di aggiornare tutti i pesi del modello simultaneamente e in modo ottimale

Where did that derivative come from?

Using the chain rule! $f(x) = u(v(x))$

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$



Derivative of the Loss: $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$

Anche con un singolo nodo in una rete neurale, ci troviamo a gestire già tre funzioni composte: la **funzione di attivazione**, la funzione di perdita (**loss function**) e l'**uscita** del nodo

Quando passiamo a reti neurali più profonde, la complessità aumenta notevolmente, poiché ogni strato aggiunge nuovi parametri e ulteriori funzioni che devono essere ottimizzate.

Per gestire questa complessità, utilizziamo uno strumento matematico chiamato **computation graph** (grafo computazionale), che ci aiuta a rappresentare e gestire i passaggi necessari per calcolare un'espressione matematica complessa.

Ogni operazione è rappresentata come un nodo del grafo e le connessioni indicano come i risultati di alcune operazioni vengono usati come input per altre

Questo grafo ci permette di scomporre anche **espressioni complesse in operazioni atomiche**

Esempio: prendiamo l'espressione $c(a + 2b)$

Possiamo scomporla in più passaggi:

- prima il calcolo di $2b$
- poi la somma $a + 2b$
- infine la moltiplicazione con c

Questi passaggi possono essere rappresentati come un grafo, dove ciascun nodo rappresenta un'operazione atomica.

Quando calcoliamo le **derivate rispetto alle variabili**, eseguiamo **il processo all'indietro** sul grafo, attraverso un algoritmo noto come **backpropagation**

Esempio: vogliamo calcolare le derivate su un computation graph dove L rappresenta la funzione di perdita (tipicamente la funzione di uscita di una rete neurale).

Il nostro obiettivo è capire come modificare i parametri (come a, b e c nell'esempio) per minimizzare L.

Il processo consiste nel calcolare le derivate parziali di L rispetto a ciascun parametro

Usando il grafo computazionale, **possiamo calcolare queste derivate** in modo efficiente, **risalendo il grafo partendo dall'output verso i singoli parametri** (per questo si parla di backpropagation)

Ciò ci permette di capire come modificare i pesi associati ai nodi della rete per minimizzare la funzione di perdita.

Se applichiamo questo concetto a una rete neurale con tre strati, significa che dobbiamo calcolare le derivate per ogni nodo della rete rispetto alla funzione di perdita.

In altre parole, per ogni nodo di uscita, calcoliamo la derivata rispetto ai pesi che influenzano l'output di quel nodo.

Riassumendo, per addestrare una rete neurale dobbiamo minimizzare la funzione di perdita (che può essere, ad esempio, la cross-entropy) calcolando le derivate dei pesi attraverso la *backpropagation*

Questo processo ci consente di ottimizzare i pesi in modo da migliorare la performance della rete

HANDS-ON

Capiremo come costruire una rete neurale da 0

Vediamo lo stack di programmazione

Si parte dalla scelta dell'hardware dato che può essere una CPU o GPU

Esistono delle applicazioni che rendono possibile applicare l'algebra dei vettori sulle CPU, mentre altre infrastrutture permettono di programmare le GPU (**esempio**: Nvidia con cura)

Sopra questo stack di basso livello, ci sono le librerie

Noi useremo tensorflow e pytorch

In realtà, tensorflow non lo useremo direttamente, ma sotto il cofano tramite keras

Keras è nato come framework a parte, senza tensorflow, perché si poteva appoggiare su altri framework, ormai in disuso

Oggi quello che si usa è l'accoppiata keras-tensorflow, infatti le nuove versioni tensorflow hanno integrato keras

ESEMPIO PRATICO: SENTIMENT ANALYSIS

Usiamo reti neurali con la rappresentazione del testo banale (bag of word), anche se il prof sconsiglia perché così non si va a sfruttare la potenza delle reti neurali

La prima cosa da fare è modificare leggermente il nostro dataset

Questo perché abbiamo un piccolo problema ad utilizzare il dataset IMDB così com'è con le reti neurali dato che il sentimento era indicato con due stringhe, positivo o negativo

Con le reti neurali non va bene perché l'output deve essere un valore numerico

Usiamo allora il `labelEncoder` che va a codificare le etichette con 0 e 1

Dopo aver letto il dataset e fatto lo split in test, validation e training set, selezioniamo le parole che compaiono almeno 30 volte

Importiamo sia il modello sequenziale (rete feed forward) sia layers da tensorflow

A questo punto, creiamo una rete la cui tipologia è decisa da noi

Creiamo un layer di input con dimensioni pari al numero di features del training set

Creiamo due layer denso con 100 nodi

Infine, un layer di un nodo che è quello di uscita

Per i layer nascosti usiamo la funzione ReLU, mentre per quello di uscita usiamo sigmoide

Ora quando facciamo `fit` aggiungiamo il parametro `batch_size`

Ogni epoca avrà 900 batch

Se sul training set l'accuracy prima o poi arriva al 100%, allora ciò significa che il modello è andato in overfitting

Se vediamo che l'accuratezza (o la precision o la recall) schizza e ciò non succede sul validation set, allora è un sintomo di overfitting

Per testare il modello si usa la funzione `evaluate`

Si può plottare un grafico per capire l'andamento di accuratezza e loss, oltre a vedere le metriche di classificazione

EARLY STOPPING

Quando notiamo che il modello funziona bene e che non ci sono più miglioramenti significativi, possiamo utilizzare una tecnica chiamata **early stopping**.

Questa tecnica si basa sull'uso di un parametro chiamato **patience**, che indica **quante epocha attendere prima di interrompere l'addestramento se non si osservano ulteriori miglioramenti nelle performance** del modello

Sebbene l'**early stopping** **NON prevenga l'overfitting**, permette di ottenere il miglior modello possibile, evitando calcoli inutili e impedendo all'algoritmo di continuare l'addestramento per troppo tempo senza miglioramenti.

Per implementare l'early stopping, dobbiamo salvare i **checkpoint** del modello.

Ogni tot batch o epoch, possiamo salvare su un file lo stato corrente del modello.

Questo si fa utilizzando le callback di Keras, in particolare la classe **ModelCheckpoint**, che è una callback che viene chiamata automaticamente durante il processo di training.

Questa callback salva il modello corrente solo se è migliore di quello precedentemente salvato

Le callback vanno passate al metodo `fit()` durante l'addestramento.

Ad **esempio**, oltre a **ModelCheckpoint**, possiamo utilizzare la callback **EarlyStopping** per evitare di sprecare cicli CPU inutili.

I parametri principali di **EarlyStopping** sono:

- **patience**: quante epocha aspettare prima di fermare l'addestramento se non ci sono miglioramenti.
- **restore_best_weights**: se impostato su `True`, al termine dell'addestramento i pesi del modello verranno automaticamente ripristinati ai migliori valori ottenuti durante l'addestramento.

Dopo aver applicato early stopping e salvato i checkpoint, possiamo caricare facilmente il modello migliore per utilizzarlo in fase di inferenza.

Dropout

Una **tecnica essenziale per prevenire l'overfitting** è il **dropout**.

Il dropout consiste nell'**inserire, durante l'addestramento**, dei **layer fittizi** nella rete che **"disattivano" casualmente alcuni neuroni in ogni batch**

In altre parole, **con una certa probabilità, alcuni neuroni NON vengono addestrati** durante un passaggio (vengono "droppati"), il che implica che i loro pesi non vengono aggiornati.

Il **concepto chiave** è che non significa che un neurone non verrà mai addestrato, ma semplicemente che, **per ogni step** di addestramento, **c'è una probabilità che un neurone NON venga utilizzato**

Questo **impedisce alla rete di dipendere troppo da neuroni specifici**, forzandola a generalizzare meglio e riducendo il rischio di overfitting

Quando costruiamo una rete neurale con **dropout**, dobbiamo **inserire esplicitamente un layer** di tipo **Dropout**, specificando la probabilità di dropout come parametro

È importante notare che il dropout viene utilizzato **solo** durante la fase di **training**, infatti in fase di test, i layer di dropout sono disattivati

Un **segnaletico di overfitting** è quando le **curve di training e validation** **NON** seguono un andamento **parallelo**. Se vediamo che le performance sul training migliorano significativamente, mentre quelle sulla validation non migliorano o peggiorano, potrebbe essere segno di overfitting.

Miglioramento del modello

Ci sono diversi metodi per ottimizzare una rete neurale profonda e ottenere risultati migliori. Tuttavia, non bisogna aspettarsi miracoli: anche con una rete più profonda, i miglioramenti potrebbero essere marginali

Per migliorare davvero le prestazioni, dobbiamo imparare a fare tre cose fondamentali:

- **Rappresentare meglio gli input:** la qualità dei dati e la loro rappresentazione giocano un ruolo cruciale nelle prestazioni del modello.
- **Calibrare automaticamente gli iperparametri:** l'ottimizzazione automatica degli iperparametri della rete, come il learning rate, la probabilità di dropout o il numero di unità nei layer nascosti, può migliorare sensibilmente il modello.
- **Progettare nuove architetture di rete:** sperimentare con diverse architetture, come le **reti ricorrenti** (RNN) o altre reti che tengono conto dello stato temporale o di altre caratteristiche, può offrire miglioramenti sostanziali, a seconda del tipo di problema affrontato.

▼ Lezione 20 - 16/05/2024

Oggi rappresentiamo le parole di una frase nell'ordine in cui appaiono.

Per farlo, costruiamo l'ingresso della rete neurale, che è di fatto un **vettore di embedding**.

Ogni parola della frase è trasformata in un vettore di embedding, ma questa rappresentazione ha un problema: l'**input della rete neurale deve avere dimensione**

costante. Di conseguenza, la frase deve avere una lunghezza fissa

Per risolvere questo problema, possiamo:

- **troncare** la frase se è troppo lunga;
- **effettuare padding**, cioè aggiungere dei token finti quando la frase è troppo corta

Questo approccio viene utilizzato anche nei modelli LLM.

Un modello di embedding può essere addestrato su una grande quantità di documenti. Quando diamo un vettore di embedding come input a una rete neurale, questi embedding vengono **ottimizzati** man mano che la rete si addestra. In alternativa, possiamo utilizzare embedding **pre-addestrati**.

Per **facilitare la gestione dei vettori di embedding** nella rete, spesso si aggiunge uno **strato fittizio** per "appiattire" i **vettori di input**, rendendoli **compatibili** con la rete.

La rete neurale deve ricevere un input di dimensioni realistiche.

Ci sono due strategie principali per farlo:

1. creare una rete neurale dove l'**input è rappresentato da un vettore di embedding**
2. **stabilire una lunghezza massima per le frasi**, utilizzando padding o troncamenti

Inizialmente, rappresentiamo le parole del nostro set di addestramento con **numeri interi**. Ogni parola viene associata a un ID numerico, e ogni frase diventa una lista di numeri. Ad esempio, la frase "John is going to the bus stop" diventa una lista di ID numerici

Padding: Per convenzione, utilizziamo lo 0 come **special ID** per il padding.

Token OOV (Out of Vocabulary): se durante il test una frase contiene parole che non erano presenti nel set di addestramento, queste parole vengono contrassegnate con un token speciale.

Inoltre, possiamo impostare una **dimensione massima del vocabolario**, rappresentando con numeri solo le parole più frequenti e contrassegnando le altre come OOV.

Tuttavia, se il vocabolario del set di addestramento è poco rappresentativo di quello del set di test, potremmo avere molte parole segnate come OOV.

Per evitare questo problema, possiamo usare un modello di embedding pre-addestrato oppure addestrarne uno specifico. Alcuni approcci più sofisticati possono anche comprendere il significato dei token OOV in base al contesto in cui appaiono.

Tutto il processo di trasformazione delle frasi viene eseguito da un **tokenizer**

Ad **esempio**, possiamo utilizzare il tokenizer predefinito di Keras, che effettua:

- lowercasing (trasformazione delle parole in minuscolo);
- rimozione di caratteri speciali;
- codifica delle parole come interi;
- gestione di un vocabolario limitato e dei token OOV.

Ad **esempio**, possiamo specificare una dimensione del vocabolario di 30.000 token, considerando solo i 30.000 termini più frequenti nei documenti di addestramento.

Le **reti generative utilizzano il tokenizzatore al contrario**: producono output numerici e ricostruiscono la frase originale a partire da questi numeri.

Ad **esempio**, ChatGPT effettua un **troncamento** del contesto man mano che si prosegue una conversazione, dimenticando parte delle informazioni precedenti.

Esempio di pipeline completa

1. **Importiamo il dataset** senza effettuare operazioni di stemming, poiché utilizziamo embedding. L'unica pulizia che facciamo è la rimozione dell'HTML.
2. **Suddividiamo il dataset** in training, validation e test set.
3. **Applichiamo il tokenizer** ai tre set separatamente e successivamente effettuiamo il padding delle frasi.
4. **Rappresentiamo il modello**: inizializziamo il tokenizer, lo applichiamo al training set e poi utilizziamo `fit` per costruire il vocabolario.

Le parole che non sono tra le più frequenti vengono contrassegnate come OOV.

Costruzione della rete neurale

La rete neurale che costruiamo è di tipo **sequenziale**

Il **primo layer** è uno di tipo **embedding**, dove ogni parola è rappresentata come un vettore.

Bisogna specificare la dimensione del vocabolario.

Prima di applicare i layer successivi, dopo il layer di embedding, inseriamo uno **strato di appiattimento** (flattening), poiché il layer di embedding è una matrice, non un layer denso. Infine, l'**ultimo layer** è una funzione di attivazione **sigmoide**.

Dopo aver creato il tokenizer, facciamo il **fit** del tokenizer sul training set, che equivale alla creazione del vocabolario. Tutte le parole che non appartengono alle 40.000 più frequenti vengono considerate OOV.

Il layer di embedding ha dei **pesi**, che vengono ottimizzati durante l'addestramento.

Quando addestriamo una rete neurale che ha come primo layer uno di embedding, stiamo facendo due cose:

1. **addestramento del layer di embedding** per codificare la semantica delle parole.
2. **calibrazione dei pesi** nella seconda parte della rete per il compito specifico (ad esempio, classificazione).

One-Hot Encoding e funzione di loss

Infine, applichiamo il **One-Hot Encoding** sul training set e lo trasformiamo anche sul validation e test set.

La funzione di loss che utilizziamo è la **categorical_crossentropy**, più indicata quando l'output non è binario e si usa la funzione di attivazione **softmax**.

Quando abbiamo un vettore **softmax**, per ottenere la posizione del valore massimo possiamo usare una funzione di **NumPy** che restituisce l'indice della posizione del valore massimo.

In questo modo, possiamo convertire l'output della softmax nella classe corrispondente alla posizione del massimo.

USING A PRE TRAINED EMBEDDING

Vediamo ora come evitare di addestrare gli embedding durante il training della rete neurale

Conviene farlo quando:

- il **dataset** è abbastanza **piccolo**, rendendo difficile addestrare nuovi embedding.
- il **dominio** di applicazione è simile a quello utilizzato per addestrare l'embedding pre-addestrato

Costruiamo quindi una rete neurale dove il **layer di embedding** è **freezato** (cioè, non aggiornabile). Questo rende l'addestramento più veloce, perché solo gli altri layer della rete verranno addestrati.

Supponiamo di avere una parola `w`, associata all'ID numerico `i`.

Se la parola `w` esiste nel vocabolario dell'embedding pre-addestrato (ad esempio, **GloVe**), popoleremo l'**i-esima riga** della matrice di embedding con il vettore associato.

Se la parola non esiste, lasciamo la riga vuota o riempita con zeri.

Procediamo con la costruzione del **training set** come fatto in precedenza.

Il pezzo chiave del codice è questo qui

```
import gensim.downloader
glove_vectors = gensim.downloader.load('glove-wiki-gigaword-100')

EMBEDDING_SIZE=glove_vectors.vector_size
voc_len=len(glove_vectors.key_to_index)

embedding_matrix = np.zeros((voc_len, EMBEDDING_SIZE))
for word, i in tokenizer.word_index.items():
    if word in glove_vectors:
        embedding_vector = glove_vectors[word]
        embedding_matrix[i] = embedding_vector
```

La matrice di embedding iniziale è una matrice di **zeri** con dimensioni `voc_len × embedding_size`, dove `voc_len` è la lunghezza del vocabolario e `embedding_size` la dimensione dei vettori di embedding

Popolamento della matrice di embedding

1. **Otteniamo l'insieme delle parole** che il tokenizer ha estratto dal training set.

2. `word_index.items()` restituisce una lista di tuple, dove ogni tupla è composta dalla chiave (la parola) e il valore (il corrispondente ID numerico).
3. Facciamo un ciclo sulle parole e sugli ID associati.
4. Se la parola è presente nel vocabolario dell'embedding pre-addestrato (ad esempio GloVe), recuperiamo il **vettore di embedding** corrispondente e lo aggiungiamo nella posizione **i-esima** della nostra matrice di embedding.

Svantaggi:

- il **fine-tuning** su modelli pre-addestrati può essere più costoso, poiché richiede l'addestramento del layer di embedding se lo sblocchiamo.
- Se si fa troppo fine-tuning, si può incorrere nel fenomeno del **forgetting**, dove la rete "dimentica" ciò che ha appreso inizialmente dal modello pre-addestrato, sovrascrivendo la conoscenza precedente.

HYPERPARAMETER OPTIMIZATION

L'ottimizzazione degli **iperparametri** consiste nel trovare i valori ottimali per i parametri della rete neurale che non possono essere appresi dal modello, come il numero di nodi, il tasso di dropout, il learning rate, ecc.

Questo processo è lungo e complesso, poiché richiede di sperimentare con diverse combinazioni di valori per migliorare le prestazioni del modello.

Un modo per effettuare l'**hyperparameter tuning** è utilizzare il package `keras-tuner`, che permette di esplorare in maniera efficiente lo spazio degli iperparametri.

Creazione del modello e campionamento degli iperparametri

1. Dropout rate:

Per definire il tasso di dropout, possiamo usare la funzione

`hp.Float` del package `keras-tuner`, che consente di effettuare un campionamento lineare tra un valore minimo e massimo.

Questo ci permette di esplorare diversi valori per il dropout.

2. Numero di nodi:

Per il numero di nodi in un layer, invece, non vogliamo fare un campionamento lineare, ma utilizziamo

`hp.Choice`, che permette di specificare un insieme discreto di valori.

In questo modo, il tuner sceglie tra i valori specificati.

Costruzione del modello sequenziale

1. Layer di embedding:

Iniziamo costruendo un modello sequenziale, aggiungendo un **layer di embedding**. Questo layer trasforma le parole in vettori densi.

2. Dropout layer:

Aggiungiamo un **layer di dropout**, specificando come **dropout rate** il valore campionato precedentemente con `hp.Float`.

3. Ciclo per i layer:

Creiamo un ciclo che aggiunge più layer densi alla rete. Il numero di layer viene scelto campionando un valore intero tra un **minimo** e un **massimo** (ad esempio 1-3 layer), utilizzando una funzione di `keras-tuner`.

4. Layer denso:

All'interno del ciclo, aggiungiamo un **layer denso**, che può avere un numero variabile di nodi.

Dopo ogni layer denso, possiamo aggiungere un altro **layer di dropout** per prevenire l'overfitting.

5. Layer finale:

Alla fine, aggiungiamo un ulteriore **layer denso** con un'architettura adeguata per l'output del nostro modello (ad esempio, per una classificazione binaria potremmo usare un'unità con attivazione sigmoide).

Compilazione del modello

Una volta costruito il modello, dobbiamo **compilarlo**

Un altro iperparametro importante da ottimizzare è il **learning rate**

Invece di campionarlo linearmente, possiamo usare un campionamento su scala **logaritmica** (ad esempio, tra 0.001 e 0.1), utilizzando `hp.Float`.

Parametri del tuner

Quando costruiamo la funzione che definisce il modello per il tuning, oltre a passare parametri come `voc_len`, `embedding_size`, e `max_len`, dobbiamo passare anche gli iperparametri che vogliamo ottimizzare (come dropout rate, numero di nodi e learning rate). Questi parametri guideranno il processo di **hyperparameter tuning** gestito da `keras-tuner`

▼ Lezione 21 - 20/05/2024

USING PRE TRAINED MODEL FROM TENSORFLOW HUB

L'idea di base è che diversi **ricercatori** hanno già pre-addestrato modelli su grandi dataset e li hanno resi disponibili su **TensorFlow Hub**.

Possiamo prendere uno di questi modelli e costruire una rete neurale più complessa aggiungendo ulteriori layer sopra il modello pre-addestrato.

TensorFlow Hub fornisce vari modelli che possiamo usare in tre modi principali:

- usarli **così come sono**, senza modifiche.
- effettuare il **fine-tuning** per adattarli meglio al nostro dataset.
- **aggiungere altri layer** sopra il modello pre-addestrato per personalizzare ulteriormente l'architettura

Uno dei vantaggi delle reti neurali è proprio la loro natura **modulare**, che consente di combinare e riutilizzare componenti già esistenti.

Creazione del modello con TensorFlow Hub

Il modello pre-addestrato si può creare facilmente importandolo da **TensorFlow Hub** usando un semplice **URL**.

Per questo esempio, utilizzeremo il dataset **IMDB** (recensioni di film) e non dovremo nemmeno occuparci della **tokenizzazione**, poiché il modello pre-addestrato si occuperà di tutto.

Dobbiamo importare il package `tf_keras`, poiché la versione di Keras inclusa in **TensorFlow** non è compatibile con **TensorFlow Hub**.

A questo punto, bisogna **definire il primo layer dal modello di TensorFlow Hub**

Utilizzeremo un **hub layer** (scaricato da TensorFlow Hub) come **primo layer** della nostra rete

Questo layer prevede di ricevere in ingresso una stringa di testo, e lo specifichiamo esplicitamente con il parametro `input_shape`

Ad **esempio**, possiamo scegliere un modello di embedding che si aspetta come input stringhe di testo

Per evitare di modificare i pesi del modello pre-addestrato, possiamo **congelare il layer pre-addestrato** impostando l'attributo `trainable=False`.

In questo modo, il layer sarà utilizzato come una **black box**, senza alterare i suoi parametri interni.

Ora, dobbiamo **creare la rete neurale**

Aggiungiamo successivamente gli altri layer alla rete neurale, esattamente come fatto nelle volte precedenti. La differenza principale è che il **primo layer** sarà l'hub layer, mentre in passato aggiungevamo manualmente un layer di input e uno di embedding.

Efficienza del training

Congelando il layer pre-addestrato (`trainable=False`), il numero di parametri da addestrare si riduce drasticamente, migliorando l'efficienza della fase di **training**.

Fine-tuning

Se invece volessimo fare il **fine-tuning** dell'intero modello, impostando `trainable=True`, dovremmo addestrare anche i parametri del layer pre-addestrato.

Questo però renderebbe il processo di fitting più lento e complesso, dato l'alto numero di parametri addestrabili.

Aggiunta di layer personalizzati

Un approccio alternativo è quello di congelare il layer di embedding (pre-addestrato) e aggiungere altri layer personalizzati a valle, che possiamo addestrare senza fare il fine-tuning di tutta la rete. Questo permette di adattare meglio il modello ai nostri dati, evitando però i costi computazionali del fine-tuning completo.

TENSORBOARD

È uno strumento che permette di visualizzare in tempo reale una serie di variabili per capire lo stato di avanzamento del training.

È una sorta di dashboard. Per usarlo, bisogna inserire una callback nel processo di training che deve contenere la directory dove andare a scrivere e la frequenza di update.

```
tensorboard serve --logdir tb_dir
```

A questo punto, si può lanciare TensorBoard che avvia un server locale sulla porta 6006. Sulla board vediamo una serie di grafici.

Un altro modo per visualizzare è con i notebook su VS Code, installando un'estensione ad hoc

ADDING DIFFERENT NODE TYPES

Abbiamo creato solo reti feedforward, quindi le parole vengono analizzate in parallelo, senza catturare le relazioni tra le parole.

Per risolvere questo, possiamo utilizzare reti leggermente diverse, come **reti convolutive** (adatte per le immagini) o **reti ricorrenti**

Un nodo di una rete ricorrente prende in ingresso un input e lo stato precedente, producendo un output e un nuovo stato (quest'ultimo resta nascosto)

In questo modo, le parole vengono analizzate tenendo conto delle parole già processate.

Una rete con questi layer ricorrenti (ad esempio **GRU gated recurrent unit**) può avere due tipi di dropout: il **dropout standard** (all'ingresso) e il **recurrent dropout** (applicato allo stato). L'addestramento di queste reti è particolarmente costoso a causa del numero elevato di parametri, dovuto alle connessioni standard e a quelle di feedback.

MODELLI TRANSFROMER

Vediamo l'architettura di rete **encoder-decoder**, molto usata nei modelli generativi.

Spesso i modelli si specializzano solo sull'encoder o sul decoder.

L'obiettivo è creare un'architettura capace di capire e generare testo.

Le reti neurali viste finora hanno un'analisi limitata, poiché non tengono conto della sequenza delle parole.

I task più ambiziosi generano una sequenza di parole in uscita, con ogni parola generata tramite un softmax che produce l'ID della parola successiva. Questo processo funziona in modo ottimizzato.

Un'applicazione tipica è la **machine translation** (traduzione automatica di frasi).

Altri task sono:

- **summarization**: riassumere un documento lungo.
- **modelli di dialogo**
- **completamento di testo**

Il modello più semplice per questo è un modello **RNN**, in cui ogni parola viene processata da un nodo RNN, che tiene conto delle parole precedentemente processate.

In questo modo, un modello può prevedere le parole successive in una frase.

L'addestramento di questi modelli è semplice, poiché non servono dati etichettati.

Molti di questi modelli sono **autoregressivi**, ossia basati sugli stati precedenti.

Dopo aver accumulato conoscenza, il modello produce una parola, e la predizione successiva terrà conto di questa.

In generale, un modello transformer prende una frase in ingresso e ne produce una in uscita.

SEQ 2 SEQ MODEL

Il modello processa una parola alla volta e, man mano, costruisce uno stato.

Ogni parola viene processata sulla base delle precedenti.

Grazie alla componente **autoregressiva**, il modello inizia a generare testo accumulando stato.

SENTENCE TRANSLATION

È lo stesso principio.

Si tratta di un modello che, con una componente autoregressiva, accumula stato e traduce una frase una parola alla volta.

ENCODER/DECODER MODEL

È una prima versione di un modello **transformer**, costituito da due componenti principali:

- **Encoder**
- **Decoder**

Non tutti i modelli hanno entrambe le componenti, alcuni si specializzano

I primi modelli erano **encoder-only**, mentre i **decoder-only** vanno bene per generare testo libero

Il **contesto** è rappresentato da un **vettore di embedding** che mantiene le informazioni relative alle parole ricevute.

La grandezza del vettore dipende dalla lunghezza massima che il modello può gestire.

Man mano che le parole fluiscono attraverso gli stati, il modello accumula conoscenza.

Ogni stato fornisce contesto agli stati successivi

In un tipico modello, abbiamo più strati di encoder e decoder

Ogni **encoder** è una **RNN** che prende in ingresso una parola (rappresentata tramite embedding) e produce uno **hidden state**, passato allo stadio successivo.

Alla fine, viene prodotto un **vettore di contesto** che passa al **decoder**.

COME È FATTO OGNI ENCODER

Ogni encoder è costituito da due componenti:

- **self-attention:** Contiene nodi **GRU (gated recurrent unit)** e permette alla rete di tenere conto delle parole precedenti durante la codifica di una nuova parola.
- **Feedforward Neural Network:** È una rete di layer densi che prende in input l'uscita del self-attention e produce l'output finale

Il **layer di attention** permette al modello di prestare attenzione alle parole precedentemente processate.

PARTE DI DECODER

Il **decoder** è simile all'encoder, ma prende **in ingresso sia il contesto dell'encoder sia l'output precedente**

Qui ci sono **due layer di attention**:

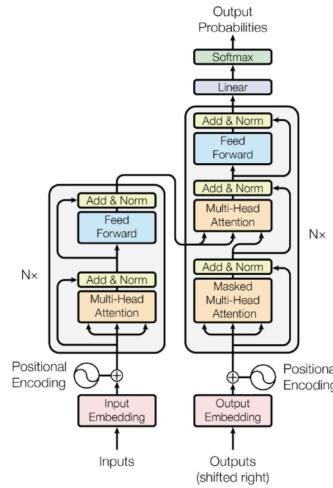
- uno tiene conto delle **parole già generate**
- l'altro **gestisce la sequenza ricevuta dall'encoder**

Il decoder riceve gli **hidden state (contesto) dall'encoder** e li elabora tramite un ulteriore layer di attention che:

- analizza gli hidden state ricevuti
- attribuisce pesi amplificandoli o deamplificandoli
- assegna uno score alle parole, con un **softmax**

Il **decoder attention** focalizza l'attenzione su parole diverse in base al contesto, dando maggiore importanza alle parole più recenti.

Trasformer Full Picture (from the original paper)



▼ Lezione 22 - 23/05/2024

Modelli Encoder-Only e Decoder-Only

- I **modelli encoder-only** sono più bravi nell'identificare le relazioni tra parole e nel produrre astrazioni rispetto alla generazione di testo.

Questi modelli sono utili per compiti come la **classificazione**.

Un

esempio di encoder-only è **BERT**, ampiamente utilizzato per compiti di **sentiment analysis**

- I **modelli decoder-only**, invece, partono da un input (chiamato **prompt**) e **generano testo** passo dopo passo, tenendo conto di ciò che è stato prodotto precedentemente. Questi modelli sono in grado di generare testo lungo e complesso.

Un

esempio famoso di decoder-only è **GPT**.

Un

limite comune di questi modelli è dato **dalla dimensione delle finestre di token**, che rappresenta la somma dei token in ingresso e in uscita. Ad **esempio**, il modello **T5** ha una finestra massima di 512 token, quindi bisogna fare attenzione.

Il

token è diverso da caratteri o parole, poiché il tokenizzatore spesso divide le parole

in più token. In GPT, mediamente, un token equivale a circa 4 caratteri o 100 token corrispondono a circa 75 parole.

Huggingface è un repository che fornisce modelli pre-addestrati e dataset utilizzati per addestrare o affinare i modelli.

I modelli **pre-addestrati** sono modelli già addestrati su determinati compiti, ma che possono essere ulteriormente raffinati (fine-tuning) per compiti specifici.

Addestramento dei Modelli

L'addestramento di un modello complesso si divide in due fasi principali:

1. **Unsupervised Denoising**: il modello impara il linguaggio utilizzando un meccanismo di **masking**. In questa fase, si **prendono delle frasi e si mascherano sistematicamente alcuni token**. Il modello deve quindi imparare a predire la parola corretta per i token mascherati. Questo processo è molto potente perché non richiede dati etichettati.
2. **Supervised Training**: questa è la fase in cui si addestra il modello per un compito specifico, come la classificazione, usando variabili dipendenti (etichette) e indipendenti (testo).

T5 è un modello **encoder-decoder** pre-addestrato in diverse lingue.

Ha già subito il fine-tuning per alcuni compiti come **traduzione e riassunto**.

Durante il fine-tuning per compiti come la traduzione, ad **esempio**, si forniscono frasi in coppie, come frasi in tedesco e il loro corrispondente in inglese.

Nel modello T5, il **prompt** (la chiave) è fondamentale perché attiva determinate parti del modello, facendo capire che deve svolgere un compito specifico. T5 è stato progettato principalmente per il **sequence-to-sequence translation**, ma può essere fine-tuned per altri task, come il **text summarization**.

Il compito di **text summarization** può essere svolto in due modi principali:

- **sommario astrattivo**: genera un sommario (o titolo) a partire da un testo astratto.
È più complesso ma produce risultati più originali.
- **sommario estrattivo**: si limita a estrarre le informazioni più rilevanti da un documento.

Nel nostro caso, vogliamo creare un titolo dato un articolo. Partiamo da un modello **T5** **pre-addestrato** in inglese e facciamo il **fine-tuning** per specializzarlo in un dominio specifico.

Per implementare il **fine-tuning** di un modello come T5, usiamo il framework **Transformers** e il supporto di **Torch**.

Tokenizzazione

Innanzitutto, dobbiamo **tokenizzare** il testo.

Per fare ciò, utilizziamo il tokenizzatore del modello scelto, che può essere recuperato da Huggingface usando il metodo `from_pretrained`.

Dopodiché, puliamo il testo, lo dividiamo in frasi, selezioniamo quelle valide (non vuote) e le aggreghiamo.

Le **label** si creano applicando di nuovo il tokenizzatore sul campo `title` degli esempi, specificando la lunghezza massima. Dall'output del tokenizzatore, prendiamo due componenti principali:

- **input_ids**: la lista di codifiche tokenizzate.
- **attention_mask**: indica quali token devono essere presi in considerazione durante il processo di attenzione del modello.

Padding e Ottimizzazione

Non facciamo padding manuale per ragioni di ottimizzazione. Usiamo invece la classe `DataCollator` offerta da `Transformers`, che gestisce il padding in modo dinamico e ottimizzato.

Valutazione del Modello Generativo

Durante il training, dobbiamo definire delle metriche specifiche per valutare le performance del modello generativo. Usiamo la funzione `compute_metrics` per stabilire le metriche.

Nella documentazione dei modelli di solito c'è un learning rate suggerito.

Facciamo un nuovo oggetto che si chiama `Trainer`

Quando salvo un modello su disco devo stare attento, perché se è un modello che è andato su CPU, una macchina che poi ha GPU potrebbe tendere a metterlo lì.

Quindi definiamo una variabile "device" che setta se usare CPU o CUDA

Per fare inferenza, è necessario che l'input e il modello siano nella stessa parte. Quindi, se il modello è su GPU, devo portare l'input lì

▼ Lezione 23 - 27/05/2024

Fine-tuning dei modelli

Ogni volta che utilizziamo un modello, dobbiamo importare sia il modello stesso che il tokenizer dallo stesso checkpoint

Il `tokenizer` produce gli `input_ids`, che rappresentano la tokenizzazione vera e propria. Applichiamo la funzione di tokenizzazione su tutto il dataset usando la funzione `map`.

La funzione delle metriche viene chiamata dalla rete neurale ad ogni step. La struttura dati `eval_pred` viene passata alla funzione stessa.

Decodifichiamo i token di questa `eval_pred` per ottenere una previsione decodificata, e facciamo lo stesso per le etichette.

Una volta decodificate, applichiamo la funzione `rouge.compute`, che prende in input la previsione e le etichette (references)

Escludiamo le predizioni nulle e calcoliamo la media delle predizioni valide, stampando infine tutte le metriche.

Metriche: ROUGE e BLEU

Per i modelli di classificazione, utilizziamo metriche come precision e recall.

Ma come si valuta un modello generativo?

Un modo semplice è l'`exact_match`, che vale 1 se la stringa prodotta è esattamente uguale all'etichetta, e 0 altrimenti.

Questo approccio, però, è limitato, soprattutto in ambiti come la generazione di riassunti

Le due metriche più popolari per validare i modelli generativi sono *ROUGE* e *BLEU*.

- **ROUGE** è una metrica simile alla recall e serve a confrontare un testo generato rispetto a uno di riferimento, calcolando quante parole del riferimento appaiono nel testo generato. Si può applicare a diversi livelli (ad esempio 1-gram, 2-grams).
- **BLEU** è usata per calcolare la precisione e introduce un fattore di penalizzazione per i testi generati troppo brevi

Quando decodifichiamo le predizioni, eliminiamo le etichette contenenti token fuori vocabolario (OOV), calcoliamo le metriche *ROUGE* e stampiamo i risultati arrotondati a 4 cifre.

Inferenza

Per fare inferenza, partiamo da un testo e creiamo un prompt.

Tokenizziamo l'input e lo spostiamo sul *device* che stiamo utilizzando (ad esempio, la GPU).

A questo punto usiamo la funzione `model.generate()`, che accetta vari parametri, tra cui

`max_new_token` e `num_beams`:

- **num_beams** ci permette di generare più soluzioni parallele
Il modello fa previsioni parola per parola, generando ogni volta la parola più probabile. Con un numero di beams maggiore, produce più frasi parallele con diverse confidenze.
- **temperatura** è un parametro che influisce sulla creatività del modello
Con temperatura bassa, il modello è meno creativo e più deterministico. Con una temperatura alta, il modello diventa più creativo, ma c'è il rischio di "allucinazioni", ossia di generare risposte non realistiche
- **repetition_penalty** penalizza il modello se tende a ripetere parole o intere frasi.

Large Language Models (LLM)

Un LLM è un modello di linguaggio, basato su deep learning e architettura transformer.

Si definisce LLM un modello che ha almeno un miliardo di parametri, anche se non esiste una convenzione universale.

Ci sono modelli specializzati per il codice, come *Code Llama* e *Codex* (il modello dietro GitHub Copilot)

L'**addestramento** di un LLM può essere fatto in due modi:

- addestrare il modello da zero (costoso e raramente necessario).
- fare un fine-tuning di un modello pre-addestrato (più conveniente).

Il fine-tuning può essere incrementale

Ad **esempio**, si può addestrare un modello prima su un task simile e poi specializzarlo. Tuttavia, alcuni LLM sono talmente grandi che non è possibile fare fine-tuning.

Zero-shot e miglioramento dei prompt

In uno scenario zero-shot, si utilizza un modello senza fare alcun addestramento specifico, semplicemente inviando un prompt al modello.

Per migliorare i prompt, possiamo fornire:

1. **Ruolo**: dare al modello un ruolo specifico, come "esperto Python" o "ingegnere junior".
2. **Informazioni contestuali**.
3. **Esempi**, usando un approccio *few-shot*.
4. **Formato dell'output** desiderato.

Modelli RAG

Un modello RAG (Retriever-Augmented Generation) combina due fasi:

1. **Retriever**: è un motore di information retrieval che usa il prompt per recuperare documenti rilevanti, costruendo una query.
2. **Generator**: usa i risultati del retriever per generare una risposta che non si basa solo sulla conoscenza interna del modello, ma anche sui dati recuperati

I modelli RAG sono utili quando non si vuole esporre i propri dati a modelli esterni.

▼ Lezione 24 - 30/05/2024

RAG consente a un LLM (Large Language Model) di sfruttare una conoscenza di dominio specifica che il modello potrebbe non avere, senza necessità di riaddestramento o fine-tuning. L'idea principale è fornire al modello maggiori informazioni contestuali.

Si parte da un prompt e, tramite un modulo di retrieval, si recuperano documenti pertinenti.

Questi documenti forniscono al modello la conoscenza necessaria per generare una risposta accurata e contestualizzata.

Inizialmente concepito per ambiti industriali, RAG oggi è utilizzato anche nel web.

LoRa (Low-Rank Adaptation)

LoRa è un'architettura usata per eseguire un fine-tuning efficiente su modelli LLM senza doverli riaddestrare completamente.

Invece di modificare tutti i pesi del modello pre-addestrato W_0 , vengono introdotte due matrici aggiuntive, A e B, che rappresentano cambiamenti incrementali.

Il processo di fine-tuning si concentra sull'addestramento delle matrici A e B, che sono relativamente piccole.

Al termine, queste matrici vengono combinate con W_0 per ottenere il modello fine-tuned, ma senza modificare i pesi originali del modello pre-addestrato.

In pratica, dato un input x, l'uscita del modello viene calcolata come:

$$h = W_0 \cdot x + B \cdot A \cdot x$$

Utilizzo di LLM all'avanguardia

Come possiamo utilizzare i LLM nei nostri sistemi? Ecco alcuni **esempi**:

1. **GPT-4**: è un modello potente, ma è a pagamento. Richiede di specificare chiaramente il contesto o i ruoli nella richiesta per ottenere le risposte migliori.
2. **Ollama**: un'alternativa più accessibile, che consente di usare modelli relativamente grandi su CPU.
Ollama funziona utilizzando versioni *quantizzate* degli LLM, riducendo la precisione dei parametri (non lavorano con

variabili a 16 bit), ma permettendo comunque prestazioni elevate su macchine meno potenti.

Si installa come servizio sulla macchina e può essere usato come chat o come web server.

Quando si usa **Huggingface**, è necessario gestire manualmente diversi aspetti del modello, mentre con Ollama tutto viene gestito come un servizio.

Vediamo come lavorare con due modelli diversi: **GEMMA** e **LLAMA**.

GEMMA

GEMMA funziona in modo limitato se eseguito su CPU, mostrando prestazioni subottimali

Un problema che emerge è legato alla

repetition penalty: se la si imposta a un valore troppo alto, il modello tende a generare output sconnessi e non coerenti.

Inoltre, l'output inizia spesso ripetendo la domanda che è stata fatta.

Per ottenere prestazioni migliori, si consiglia l'uso su GPU.

LLAMA

Con **LLAMA**, il processo è simile, ma con qualche differenza

LLAMA viene fatto girare su GPU, quindi è importante che l'input e il modello si trovino sullo stesso dispositivo (CPU o GPU). Se il modello si trova su GPU, anche l'input deve essere trasferito lì.

Con **LLAMA3**, la preparazione dell'input è leggermente diversa, e il formato dei messaggi segue uno schema simile a quello di GPT, quindi va tenuto in considerazione.

La tokenizzazione deve essere fatta in modo che corrisponda al dispositivo su cui il modello è caricato (impostando il tokenizzatore su `model.device`).

Per generare l'output, si utilizza la funzione `generate()`, a cui vanno passati anche i **terminatori** (caratteri che indicano la fine del testo), altrimenti il modello potrebbe continuare a generare testo indefinitamente, causando confusione o errori.

ISSUES WITH LLMs

Alcuni classici problemi degli LLMs sono:

- bias, cioè alcuni modelli potrebbero discriminare
- diritti d'autore, cioè è difficile stabilire la fonte su cui si è basato un modello per fornire una risposta