Relazione

Questo progetto si divide in cinque file (Preprocessing, Processing, prova, Data\_analisys e util ) e una directory (Datasets).

# Datasets

Partendo dalla directory troviamo al suo interno quattro file rappresentanti i nostri dataset, tre disponibili dal sito Kaggle da cui abbiamo preso la sfida, uno di nostra creazione per semplificare il processo di predizione. Tutti i file sono informato tsv (tab-separeted-values) e quindi di facile lettura e comprensione:

* **Labled Train Data**: il nome è al quanto esplicativo, infatti questo file è il dataset che viene usato per il train del modello. Si compone di 25000 righe e tre colonne:
  + Id: la colonna che associa ad ogni recensione uno specifico id, irrilevante per quanto riguarda il progetto in se.
  + Review: il cuore del nostro train set, ovvero la X, questa colonna contiene tutte le 25000 recensioni di film prese dal sito IMBD contenenti alcuni pezzi di linguaggio HTML.
  + Sentiment: ciò che rende la classificazione supervised, infatti questa è la nostra Y nel momento dell’allenamento del modello. Il suo valore è binario ed è 1 se la recensione è positiva, 0 altrimenti.
* **Unlabled Train Data**: questo file si compone di 50000 righe e il suo formato è simile a quello descritto in precendeza. L’unica differenza sta nel fatto che non è presente la colonna “Sentiment”, ovvero la Y. Abbiamo comunque usufruito di questo dataset nel momento della trasformazione delle recensioni da stringhe a vettori numerici.
* **Test Data**: anche qui il nome dice tutto. Il file si compone di 25000 righe e due colonne, una contenete le recensioni l’altra l’id.
* **Test Data Labled**: questo è il dataset da noi creato. Il motivo che sta alla base di questo passo è la mancanza della colonna “Sentiment” nel test dataset fornito da Kaggle. Infatti il sito fornisce la possibilità di inviare la propria predizione per ottenere il risultato, ma questa viene limitata per sei volte nell’arco di 24 ore. Cercando su google ho scoperto che nell’id del Test Dataset è presente il voto in decimi correlato alla recensione, e dato che la sfida fornisce il criterio di valutazione ( un voto minore di 5 indica una recensione negativa mentre uno maggiore di 7 una positiva), abbiamo creato una piccola funzione ausiliaria che ci ha fornito la colonna “Sentiment” facilitando il processo di stima dell’errore.

# Preprocessing

Qui è dove i dati vengono trattai per permettere una migliore classificazione.

Le funzioni principali sono tre e vengono chiamate nell’ordine nelle quali adesso le illustreremo.

**Sentence polishing**

Il compito di questa funzione è quello di ripulire le recensioni da tutta una serie di agenti “inquinanti” che possono traviare il classificatore.

La funzione prende come input una lista di stringe, in questo caso tutte le recensioni, e vi applica cinque tipi di filtraggio tramite l’uso della list comprehension e di funzioni provenienti dalla libreria [Genism](https://radimrehurek.com/gensim/):

* **Rimozione dei tag di HTML**: come accennato prima le recensioni vengono prese direttamente dal sito HTML e per ciò portano con se alcuni pezzi di linguaggio HTML. Qui abbiamo usato la funzione strip\_tags che ha esattamente il compito di eliminare i tag HTML.
* **Rimozione delle stop words**: le stop words sono parole presenti in tutte le lingue la cui ripetizione nelle frasi è elevata ma non sono di vitale importanza per il training di un modello, infatti possono essere anche causa di stime molto basse nella predizione poiché la loro frequenza oscura parole di maggior rilevanza. Alcuni esempi di stop words possono essere trovati [qui](http://www.ranks.nl/stopwords).
* **Rimozione della punteggiatura**: come per le stop words, anche la punteggiatura può influire negativamente sul training del modello, per questo abbiamo deciso di rimuoverla.
* **Rimozione dei caratteri non alfa numerici**: essendo le recensioni scritte da persone possono essere presenti dei caratteri non alfanumerici. La frequenza di questi caratteri è molto bassa, quindi non rappresentano un vero e proprio ostacolo per i modelli ma sono comunque stati tolti per questioni di ottimizzazione di memoria.
* **Rimozione di frasi vuote**: è possibile che dopo i precedenti filtraggi alcuni elementi della lista siano vuoti, ciò genererebbe degli errori al tempo di esecuzione e quindi le abbiamo eliminati.

Dopo questa serie di rimozioni la lista ha perso circa un terzo dei suoi elementi, garantendo un miglioramento in termini di memoria utilizzata, tempo di esecuzione e precisione nella predizione.

**String2VecTFIDF**

Una volta puliti i datasets abbiamo bisogno di trasformare le parole in vettori, per questo compito ci siamo avvalsi della funzione [TfidfVecorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) fornita dalla libreria sklearn.

La teoria alla base di questo algoritmo di vettorializzazione è relativamente semplice.

Quando ho un vasto insieme di stringhe alcune parole, per esempio le stop words, compaiono molto frequentemente. Utilizzando un normale algoritmo basato sulla bag of words, i pesi associati a queste parole maschererebbero la vera importanza di altre parole meno frequenti ma di più significato. Per questo il TF-IDF (term-frequency-inverse-document-frequency) utilizza due termini:

* Il **term frequency**: un intero che rappresenta il numero di volte che uno specifico termine compare in un documento (in questo caso in una recensione)
* **L’inverse document frequency**: semplicemente text{idf}(t) = log{\frac{1 + n_d}{1+\text{df}(d,t)}} + 1 , dove rappresenta il numero delle recensioni e df(d,t) è il numero di recensioni che contengono il termine t.

Questi due termini vengono moltiplicati e il loro prodotto è poi normalizzato secondo la norma euclidea.

Nel nostro caso sono presenti anche dei termini all’interno della funzione che adesso andremo a spiegare:

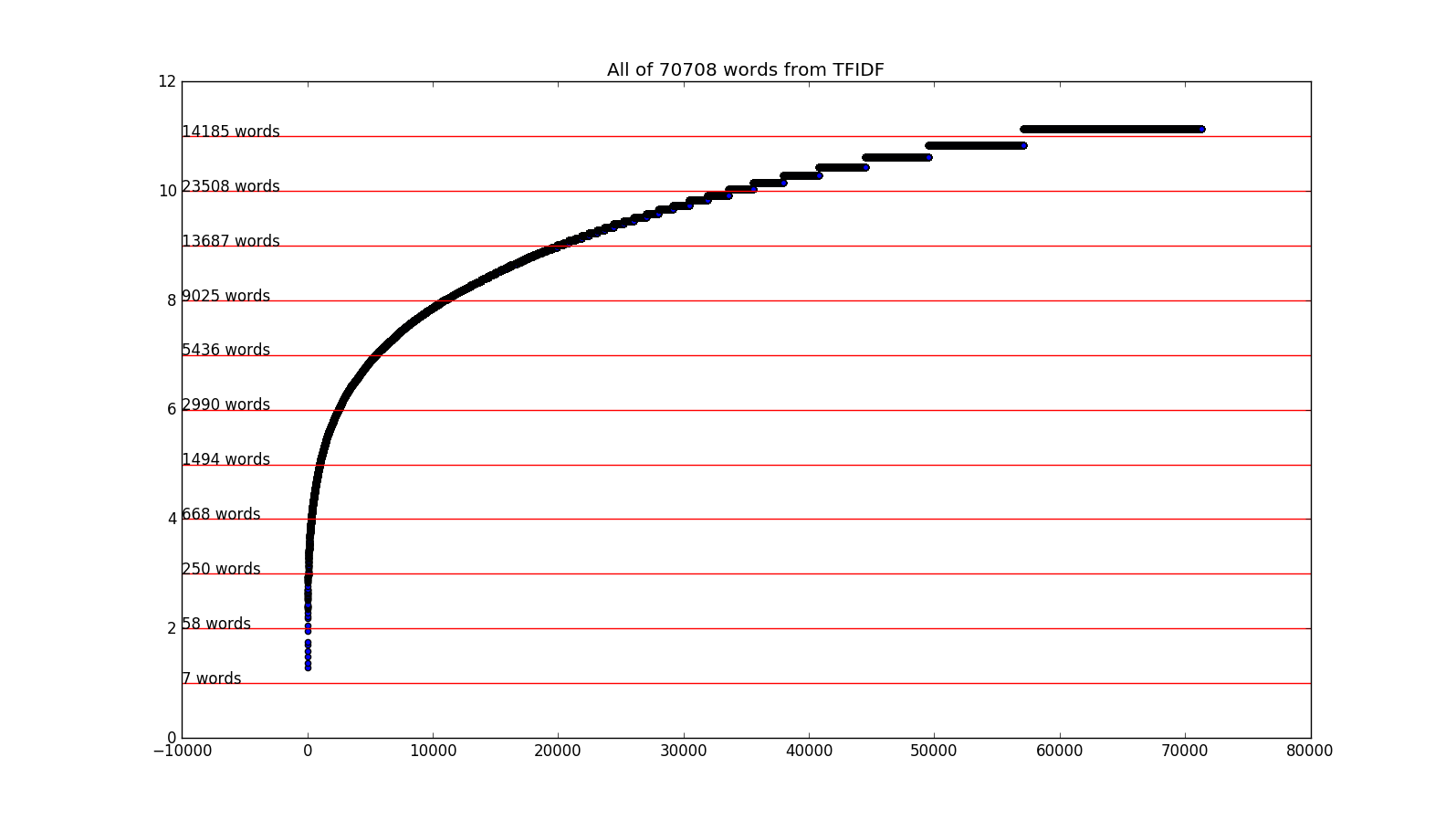
* **Min\_df**: alcune parole hanno una frequenza cosi bassa all’interno della recensione che è possibile ignorarle, per questo abbiamo settato questo parametro a 2, cosi da eliminare tutte le parole che si ripetono meno di tre volte.
* **Max\_df**: semmai la nostra precedente pulizia delle stop words possa aver tralasciato qualcosa, entra in gioco questo parametro settato a 0.96. In pratica ordino alla funzione di ignorare le parole che si ripetono per più del 96% all’interno della recensione. Un esempio è dato da una recensione molto corta la quale contiene solamente ‘XD’.
* **Sublinear\_tf**: è una semplice sostituzione del parametro Term frequency, che diventa 1+log(tf).
* **Max\_features**: essendo il dataset composto da oltre 70000 parole abbiamo deciso di prendere solo le 20000 più rilevanti per evitare di portare il tempo di esecuzione oltre i 10 minuti.
* **Strip\_accents**: rimpiazza le lettere accentate con normali caratteri ascii, questo è stato possibile poiché la lingua inglese non presenta profonde differenze tra parole accentate e non, a differenza di quella italiana.

Dopo l’inizializzazione si passa al fittaggio del dataset e qui è stato di fondamentale importanza l’uso dell’Unlabled Train Data che ci ha permesso di aggiungere al dizionario creato dalla funzione più del 50% dei termini, aumentando l’accuratezza delle predizioni del 3% in media.

Finito il fittaggio siamo passati all’effettiva trasformazione del dataset dal domino delle stringhe a quello numerico tramite il metodo “transform” applicato al Train Labled Data e al Test Data.

In fine la funzione ritorna il Train e il Test datasets vettorializzati insieme all’oggetto TfidfVectorizer stesso (solamente per motivi di debug).

Tramite la funzione “plot\_vector” nel file Data\_analisys siamo riusciti a plottare il risultato della vettorializzazione:

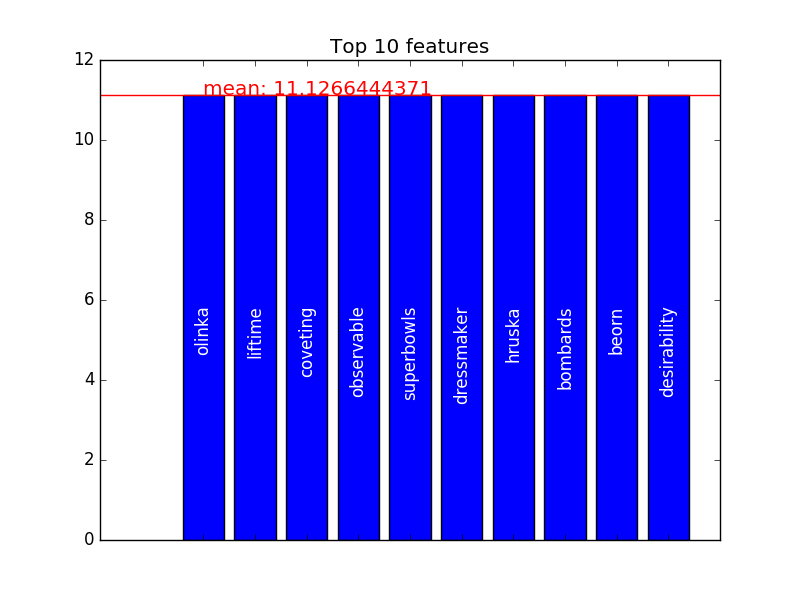


Sull’asse delle ascisse sono ripostati il numero delle parole del dizionario creato mentre sull’asse delle orinate il peso associato. Per una migliore comprensione del grafico abbiamo diviso l’asse y con una serie di rette in rosso e un numero che rappresenta le parole che hanno ricevuto un punteggio nel range di valori unitario. E’ interessante notare come più del 50% delle parole sia compreso nei range 9-10, 10-11 e >11, da qui la scelta di impostare il massimo numero delle features a 20000.

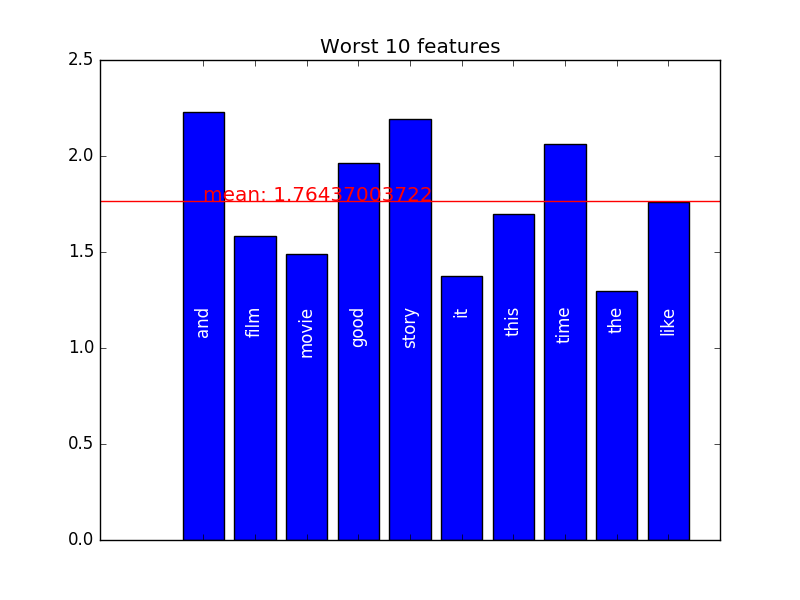
**Dimensionality\_reductionKB**

Dopo aver trasformato le stringhe in vettori, ci siamo posti l’obbiettivo di ridurre ulteriormente il numero di features. Il problema con la funzione precedente è che questa assegnava alle parole un peso in base alla frequenza all’interno del dataset, ma non teneva conto del significato in relazione alla predizione, anche perché la funzione non prende in input le Y.

Infatti abbiamo plottato le migliori 10 features e le peggiori 20 per il TFIDF tramite la funzione plot\_top\_n\_words nel file Data\_analisys e abbiamo ottenuto i seguenti risultati:



Come possiamo vedere le parole associati ai pesi maggiori non hanno particolare comunanza semantica, né interesse per valutare una recensione.

Mentre nelle peggiori 20 sono presenti sia stop words che parole molto interessanti quali “good” e “like”.

Per questo abbiamo usato la funzione [SelectKBest](http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html) unito alla stima [Chi2](http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html#sklearn.feature_selection.chi2) della libreria sklearn.

Adesso passiamo a spiegare la teoria e il funzionamento dietro queste due funzioni.

**Chi2**

Prima di entrare nel particolare ricordiamo a caratteri generali che il chi2 viene utilizzato per misurare la dipendenza tra variabili stocastiche, in questo caso tra una feature e le Y, cosi facendo eliminiamo le features che risultano indipendenti dalle Y e che quindi non hanno importanza nella predizione.

Abbiamo scelto la funzione di stima chi2 per due ragioni fondamentali:

* Il nostro dataset è rappresentato da una matrice sparsa, ovvero da una matrice con una maggioranza di zeri. L’opposto di una matrice sparsa è una matrice densa, in cui vi è un’alta probabilità di trovare una diversa combinazione di features per ogni riga. La classificazione, o più in generale il train su di una matrice densa rende il fitting estremamente lento e complicato, per questo abbiamo optato per la funzione chi2 che effettua la stima delle features senza rendere la matrice densa.
* L’altra ragione, scontata, ma di fondamentale importanza, risiede nelle Y. Infatti queste sono categoriche (e binarie), se cosi non fosse avremmo a che fare con un problema di regressione intrattabile con il test del chi2.

Volendo entrare di più nel dettaglio il chi2 si basa sull’ipotesi nulla, ovvero date due variabili categoriche questa sono assunte essere indipendenti tra loro finche non viene dimostrato il contrario.

Per smentire l’ipotesi nulla il procedimento chi2 è il seguente:

* Innanzitutto si calcola il cosi detto “Expected Count” [EC] per ogni cella del dataset:
  + RT= sarà la somma di tutte le features di una riga
  + CT= è la somma di tutte le righe di una features
  + SS= è la somma su tutte le righe e tutte le colonne
  + La formula applicata è EC=(RT\*CT)/SS
* Quindi l’EC rappresenta il valore che la cella avrebbe se le due variabili fossero davvero indipendenti. Il procedimento chi2 ora passa a rispondere alla seguente domanda: “La differenza tra L’EC e i valori delle celle è cosi importante da confermare una interdipendenza delle due variabili?”.
* Introducendo le seguenti variabili:
  + : valore originale della cella i-esima
  + : valore dell’EC della cella i-esima
  + = numero di righe
  + = numero di colonne
  + , denominato degree of freedom
  + n=numero di colonne\* numero di righe
* L’equazione del chi2 è
* Adesso bisogna calcolare il valore di P e per far ciò abbiamo bisogno del parametro DF e del livello di significatività [LS], il quale ci da una misura del livello di certezza che volgiamo ottenere dal risultato. Valori bassi di LS indicano che ci sia una bassa probabilità che i risultati siano casuali, solitamente questa parametro viene impostato a 0.05, ovvero il 5% di possibilità che l’evento sia riproducibile in un random sampling.
* L’ultimo passaggio sta nel cercare il valore di nelle tabelle del chi2, utilizzando il DF. Se il VC è minore di 0.05 possiamo sicuramente smentire l’ipotesi nulla e quindi esiste una correlazione tra le variabili, altrimenti non possiamo rigettare l’assunzione iniziale.

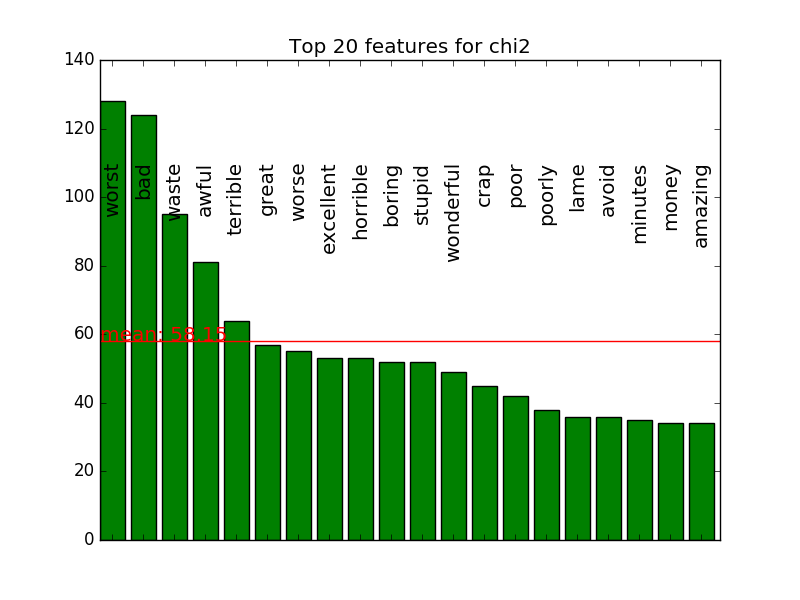
**SelectKBest**

Questa funzione non fa altro che prendere in input due parametri:

* La score function: abbiamo usato il chi2 per le ragioni precedentemente citate
* K: impostato all’85% di tutte le features presenti nel dataset, grazie al tuning, rappresenta le miglori caratteristiche da selezionare.

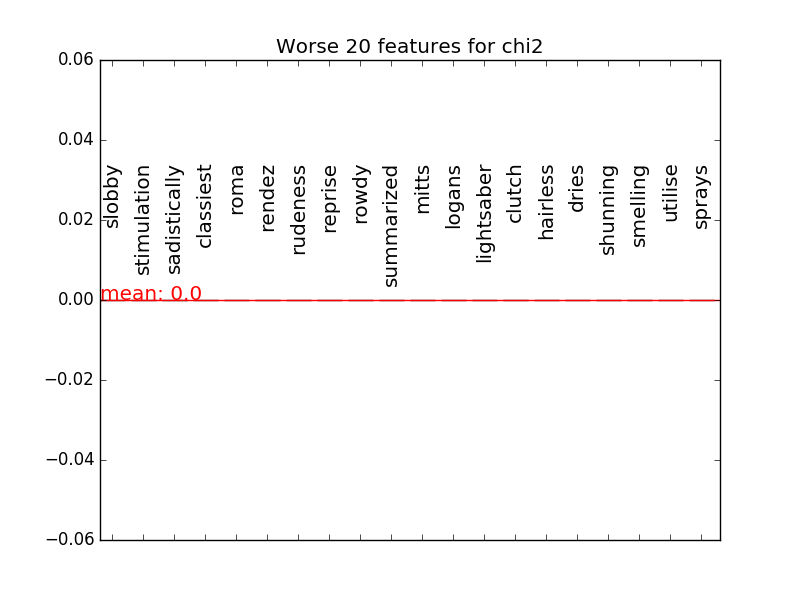
Eseguendo il metodo fit\_transform e passando come paraemtri le X e le Y la funzione esegue (per un tempo medio di 70 secondi) l’algoritmo e ritorna una matrice in cui sono presenti solo l’85% delle migliori features con pesi opportunamente ridimensionati.

Tramite la funzione plot\_chi2\_top nel file Data\_analisys abbiamo plottato le migliori 20 parole ritornate dalla funzione ottenendo i seguenti risultati:



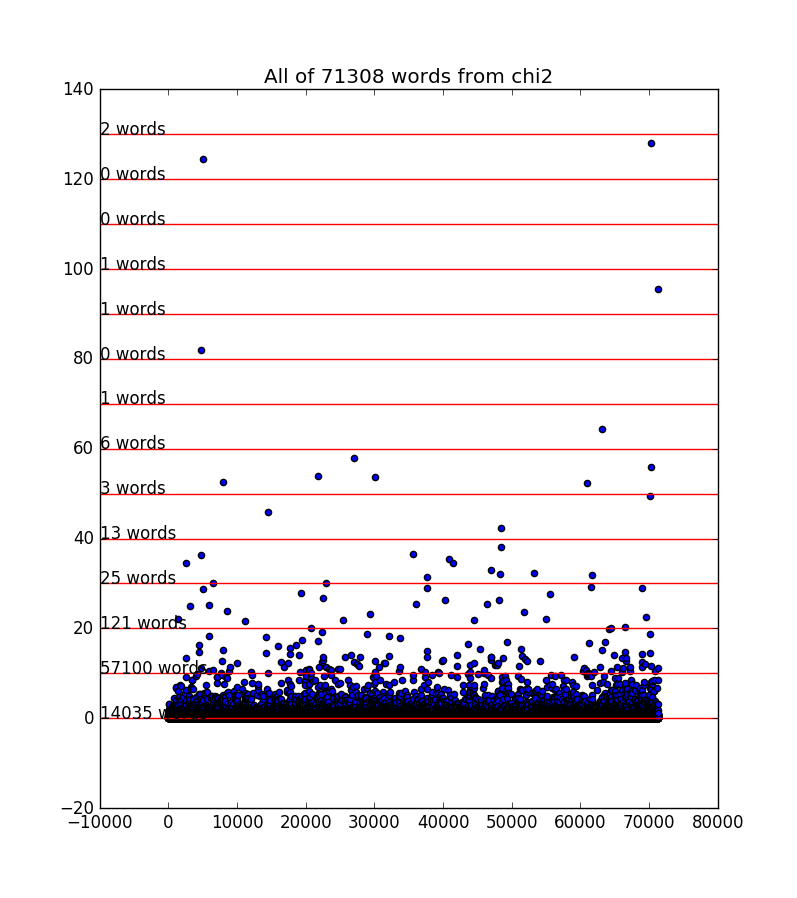
Possiamo notare come la maggior parte delle parole suonino fondamentali per l’analisi dei sentimenti in una recensione. E’ anche interessante vedere come due parole “minutes” e “money”, che per un essere umano possono essere considerate secondarie, siano comunque comprese nella top 20 per l’algoritmo.

Abbiamo anche plottato le peggiori 20 parole:



Anche qui i risultati confermano ciò che potevamo aspettarci.

Plottando tutti le parole con i relativi pesi del chi2 ci siamo accorti che la maggior parte delle features è associata ad un voto basso e solo una minima parte sono fondamentali:



**String2VecCV**

Questa funzione è molto simile a quella descritta precedentemente (string2vecTFIDF) solo che si basa sul principio della bag of words….

# Util

Questo file si compone di due semplici funzioni utilizzate per ridurre al minimo le righe di codice durante l’esecuzione del programma.

**Scoring**

Prende in input un vettore di predizioni fornito da un classificatore e mostra sullo schermo la media dell’errore tra la predizione e l’effettivo risultato del test.

**Pliosh\_TFIDF\_kbest**

Presi in input entrambi i Train datasets e il Test set esegue la vettorializzazione e il kbest come descritto precedentemente.

# Processing

Questo è il file principale di tutto il programma, qui avviene il fittaggio e la calssificazione per i vari modelli.

Le funzioni sono tre: due contengono un modello a testa e l’ultima ne contiene due.

**Forest\_classifier**

Questa funzione si avvale del metodo precedentemente citato nel file Util (Poish\_TFIDF\_Kbest) per preprocessare i dati che poi vengono passati al classificatore.

Spiegheremo adesso il funzionamento di una Random Forest.

Una Random Forest, o Decision Forest, è un meta estimatore che si basa su una moltitudine di alberi decisionali su cui viene fatto il fittaggio del dataset. La creazione di un albero è decisa in base alla migliore stima ottenuta su di un sotto insieme randomico del dataset. La Random Forest fa uso di tecniche di media durante la composizione di più alberi, questo riduce la sua varianza aumentando l’accuratezza del modello ed evitando l’over fitting.

Per capire meglio come funziona questo algoritmo dobbiamo analizzare le proprietà e le tecniche alla base di un albero di decisione.

**L’albero di decisione**

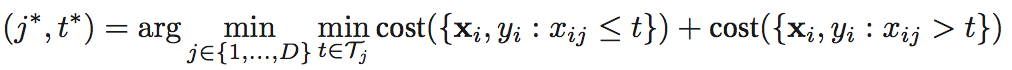
Un albero di decisione funziona in base a più divisioni paralleli lungo l’asse. Ogni divisione indica una regione diversa in cui la feature può trovarsi, queste regioni sono create in base ad una funzione base che andremo a spiegare più avanti e sono affiancate da pesi che specificano la il valore di ogni regione.

La funzione è la seguente: , in cui:

* M: è il numero di regioni dell’albero
* : sono i pesi dei rami, che determinano il valore limite della feature (esempio se abbiamo omega=2 allora la feature verrà indirizzata in un ramo se il suo valore è <2 e in un altro altrimenti)
* : è la variabile
* : è la m-esima regione

Una domanda che sorge spontanea è la seguente: Qual è la partizione ottimale per il dataset?

L’albero decisionale adotta un approccio greedy per calcolare la MLE ottimale per la divisione delle migliori features e ai loro migliori valori, la funzione è la seguente:



in cui:

* j\*: e la feature ottimale
* t\*: è il valore ottimale della feature j\*
* la prima funzione di minimo è applicata a tutte le features del dataset
* la seconda funzione di minimo è applicata a tutte le possibili soglie di valori della feature j, ottenibili ordinando tutti i valori univoci della variabile j (esempio: se la feature n-esima ha come valori [55,3,6,1,3] allora )
* il costo per la creazione di due nuovi rami, le funzioni di costo variano per problemi di calssificazione o regressione

**Funzioni di costo**

Nel caso di regressione la funzione di costo è semplicemente la somma del quadrato dell’errore tra y predetta è y reale.

Per la classificazione, invece, viene fittata un Moultinoulli, per ogni foglia dell’albero: Questo modello deve stimare la class-conditional probability nel seguente modo: , in cui D sono i dati nella foglia.

Fatto questo, il modello deve stimare l’errore di classificazione tramite diversi metodi, noi abbiamo optato per la minimizzazione dell’entropia tramite la formula: dove è la MLE per la distribuzione p(c|Xj<t) precedentemente citata.

**Bagging**

La particolarità di questi alberi, molto usati nel machine learning, sta nel fatto che tendono ad un overfitting del train set, portando ad un aumento della varianza con relativa diminuzione della media. Per ovviare a questo problema le Foreste fanno spesso uso della Bootstrap Aggregation, più comunemente denominata bagging, la quale divide il dataset in N parti randomiche in cui possono essere presenti ripetizioni delle stesse osservazioni. Dopo la divisione il modello fitta gli alberi su questi sotto insiemi del dataset e, per quanto riguarda la classificazione, combina i migliori alberi per votazione. Questo procedimento permette di diminuire la varianza nel modello senza aumentare la media, questo perché la combinazione di più alberi rende il Random Forest più resistente al rumore.

Un altro motivo per cui il modello fa uso del bagging è la correlazione tra gli alberi. Se il fittaggio avvenisse sulle stesse parti di dataset si verrebbero a creare alberi molto simili tra loro, questo porterebbe il modello a sceglierli, nella fase di avereging, non in base al voto sulla stima ma in base alla correlazione che hanno tra loro.

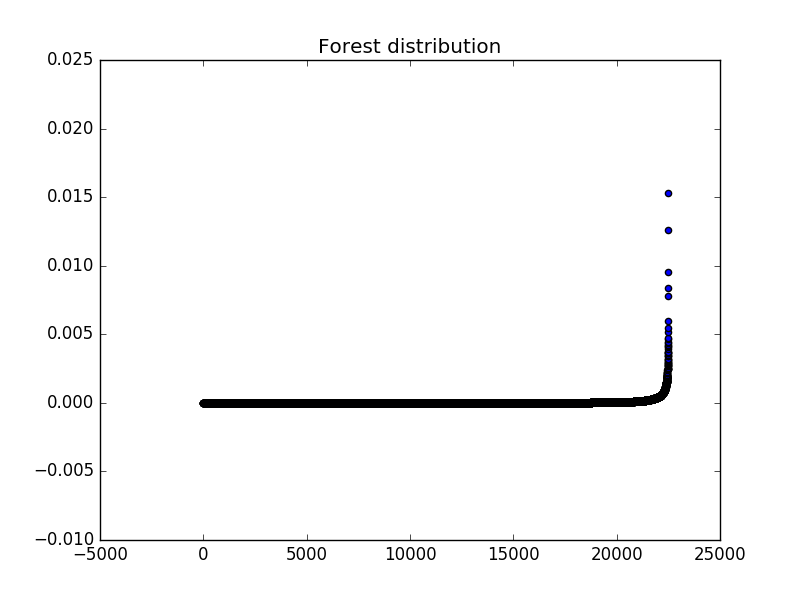
**Parametri adottati**

Per quanto riguarda i parametri che abbiamo scelto per il classificatore questi sono quattro:

* **n\_estimators**: il numero di alberi della foresta. Abbiamo impostato questo parametro a 300 poiché, dopo un po’ di tuning, ci siamo accorti che il limite massimo per l’accuratezza era di circa l’85% ed aumentare il numero di alberi non faceva altro che aumentare il tempo computazionale senza portare particolari migliorie nella predizione.
* **N\_jobs**: questo parametro permette la parallelizzazione del processo di fittaggio. Il -1 sta ad indicare che il modello usa tutti i core disponibili sulla macchina.
* **Verbose**: parametro selezionato puramente per motivi di debug, mostra l’avanzamento durante il fittaggio.
* **Criterion**: questo parametro stabilisce quel criterio usare per la stima dell’errore di classificazione, abbiamo optato per il modello Entropy descritto precedentemente.

**Grafici**

In fine abbiamo plottato i pesi che la Random Forest ha assegnato alle features:



E’ interessante notare che solo una piccola parte delle fetaures sono importanti per la random forest.

**SGD\_classifier**

Il secondo classificatore adottato è lo Stochastic Gradient Descend [SGD] che ci ha portato ad un’accuratezza di circa 88%.

L’euristica dietro il modello SGD è più complicata rispetto a quella della Random Forest, perciò apporteremo delle semplificazioni per quanto riguarda la seguente analisi.

**Teoria Generale**

Considerando una loss function , che misura il costo di una predizione [y] data il valore originale [], e una famiglia di funzioni F parametrizzate da un vettore di pesi omega, volgiamo cercare la funzione tale che questa minimizzi la loss function.

Adesso consideriamo il rischio empirico come , per minimizzare questo parametro viene usata la tecnica del gradient discent, quindi ad ogni operazione viene aggiornato il parametro omega dei pesi in base al gradiente calcolato sull’intero dataset, questo comporta due svantaggi significativi:

1. Il modello non può essere sottoposto a training online, poichè è necessario l’intero dataset per effettuare il gradiente
2. Il tempo computazionale è proporzionale alla grandezza del dataset e, per matrici sparse di grandi dimensioni come quella che viene adottata in questo programma, il training del modello impiegherebbe ore.

L’SGD si differenzia leggermente dall’approccio classico del gradiente, infatti questo viene calcolato su poche parti del dataset e il nuovo parametro omega è cosi calcolato: , in cui:

* sono una coppia presa dal dataset
* è il learning rate [LR]
* è la funzione obbiettivo, in questo caso il rischio empirico
* è il gradiente

Questo rende l’algoritmo più veloce, poichè può essere sottoposto a tecniche di parallelismo vettoriale, e più preciso.

**Learning Rate [LR]**

E’ stato dimostrato che quando il LR diminuisce gradualmente e se la funzione obbiettivo è convessa allora l’SGD converge ad un minimo locale, altrimenti la conversione avviene intorno ad un minimo locale.

La teoria ci dimostra che il migliore LR prende la forma di in cui è il più piccolo autovalore della matrice Hessiana, inoltre sovrastimare anche solo di di porta il modello a convergere molto lentamente. Nel caso si usi un termine di regolarizzazione (esattamente quello di cui abbiamo usufruito) nel training allora avremo e quindi possiamo adottare una tecnica di decrementazione nella forma .

Nel nostro caso la libreria sklearn offre diversi approcci per il calcolo del learning rate. Quello che abbiamo scelto di adottare è denotato “optimal” e viene calcolato ad ogni iterazione con la seguente formula: in cui:

* : è un parametro, chiamato costante di regolarizzazione, preimpostato a 0.0001 la quale funzionalità verrà spiegata più avanti
* : è il passo corrente nella computazione
* : è scelto secondo l’euristica proposta da Leon Bottou, in cui il parametro viene settato leggermente minore del miglior valore osservato nella porzione di dataset su cui è effettuato il gradiente.

**Termine di Regolarizzazione**

Anche in questo caso la libreria sklearn ci permette di scegliere il termine di regolarizzazione e la sua costante. Come precedentemente detto abbiamo optato per (anche chiamata Ridge Regression) con costante di default , adesso passiamo a spiegare in cosa consiste questo termine.

Il termine di regolarizzazione è un oggetto fondamentale nel machine learning, il suo scopo è quello di prevenire l’overfitting aggiungendo un parametro durante il calcolo dei pesi. La formula è per è semplicemente la somma del quadrato dei pesi per la costante: che viene sommato alla funzione obbiettivo. I vantaggi nell’usare rispetto a sono molteplici:

* possiede una soluzione analitica e per ciò la sua computazione è molto più efficiente rispetto ad .
* non inserisce sparsità nel calcolo dei pesi.

Per quanto riguarda il coefficiente di regolarizzazione questo deve essere opportunamente scelto, nel caso di un alfa troppo grande il modello potrebbe essere soggetto a underfitting, al contrario, per alfa eccessivamente piccolo, il termine di regolarizzazione risulterebbe inefficace.

**Loss Function [LF]**

Prima di tutto diamo la definizione di LF: una loss funcition è una funzione che associa dei valori di una o più variabili ad un numero reale, quindi determinando un costo associato all’evento.

La LF che abbiamo scelto per il nostro classificatore è chiamata “Modified Huber” che discende dalla più comune funzione di perdita “Hinge” con la differenza di essere più regolare.

La funzione Hinge è solitamente usata per la Support Vector Machine [SVM] nei problemi di calssificazione, la sua formula è la seguente:

in cui:

* è il vettore dei pesi
* y è la predizione del modello
* x è la variabile che si vuole predire

Una proprietà che distingue la Hinge dal resto delle funzioni di perdita è che questa è più benevola nei confronti degli errori e del rumore nel training set.

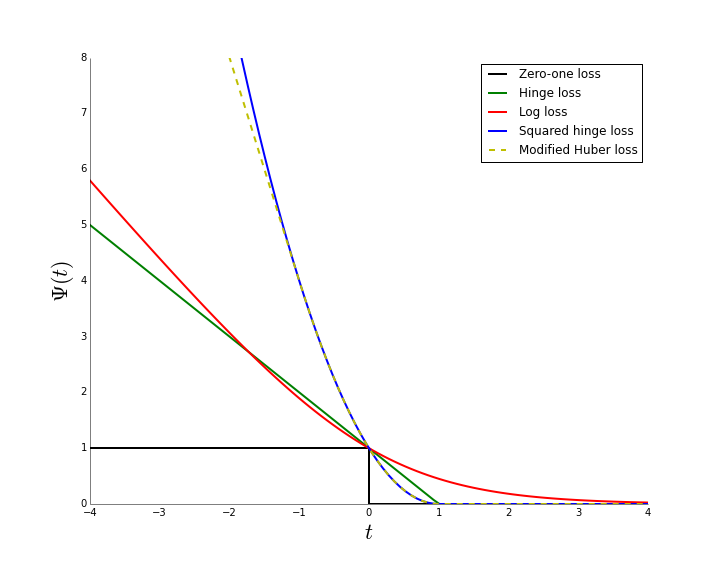
Per quanto riguarda il “Modified Huber”, questa viene usata

per problemi di classificazione binaria e ha la seguente forma (come riportato alla riga [108](https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/linear_model/sgd_fast.pyx) dell’implementazione) :

con

Possiamo notare come questa funzioni sia più “smooth” della precedente poichè divide il piano in tre parti.

Qui sotto riportiamo il grafico che caratterizza diverse funzioni di perdita tra cui Hinge e Modified Huber.



**Parametri**

Nell’ultima parte di questo sotto capitolo spiegheremo i parametri usati e le ragioni che ne stanno alla base:

* **Verbose**=1: stampa informazioni durante l’esecuzione, usato per il debug
* N\_**jobs**=-1: parallelizza il processo di fittaggio con l’ausilio di tutti i core della macchina
* **Loss**=Modified\_hiber: scelta della funzione di perdita precedentemente citata
* **Random\_state**=4: il seed per miscelazione randomica del dataset
* **N\_iter**=10: il numero di volte che il modello viene esposto al trainig dataset
* **Shuffle**=True: mischia il dataset prima di iniziare il fittaggio