Relazione

Questo progetto si divide in cinque file:

* Preprocessing
* Processing
* Runner
* Data\_analisys
* 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 ho preso la sfida, uno di mia 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 l’allenamento 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 sé.
  + 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 precedenza. L’unica differenza sta nel fatto che non è presente la colonna “Sentiment”, ovvero la Y. Ho 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 me 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. Effettuando una ricerca 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), ho creato una piccola funzione ausiliaria che ci ha fornito la colonna “Sentiment” facilitando il processo di stima dell’errore.

# Preprocessing

Questo file contiene quelle funzioni che trattano i dati, prima della classificazione, per permettere una migliore classificazione.

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

**Sentence polishing**

Il compito di questo metodo 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 HTML**: come accennato prima le recensioni vengono prese direttamente dal sito in formato HTML e per ciò portano con se alcuni stralci di questo linguaggio. Qui ho usato la funzione strip\_tags che ha esattamente il compito di eliminare questi caratteri non indispensabili.
* **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 l’allenamento 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 ho deciso di rimuoverla.
* **Rimozione dei caratteri non alfa numerici**: essendo le recensioni scritte da persone, possono essere presenti dei caratteri non alfa-numerici. La ripetizione 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 ho deciso di eliminarle.

**Stemming\_lemmatization**

Questa funzione apporta un’ulteriore modifica al dataset utilizzando due tecniche:

* **Stemming**: Il processo di stemming consiste nel ridurre le parole alla loro radice o forma base. L’algoritmo che sta alla base di questo è chiamato Snow Ball e la spiegazione completa può essere trovata [qui](http://snowball.tartarus.org/algorithms/porter/stemmer.html).
* **Lemmatization**: La lemmatizazione è il procedimento secondo il quale gruppi di parole con diverse inflessioni vengono raggruppate insieme cosi da poter essere analizzate come un singolo elemento. La differenza principale tra questi due algoritmi sta nel fatto che lo stemmer analizza una parola per volta mentre il lemmatizer le raggruppa insieme.

Per chi fosse interessato a tesare diversi stemmer e lemmatizer [qui](http://textanalysisonline.com/nltk-lancaster-stemmer) potete farlo.

Dopo questa serie di rimozioni la lista ha perso circa un terzo dei suoi elementi (esattamente il 35%), garantendo un miglioramento in termini di memoria utilizzata, tempo di esecuzione e precisione nella predizione.

**String2VecTFIDF**

Una volta puliti i datasets si è presentata la necessità di trasformare le parole in vettori, per questo compito mi sono avvalso della funzione [TfidfVecorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) fornita dalla libreria [Sklearn](http://scikit-learn.org/).

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

Nel comento in cui si ha vasto insieme di stringhe alcune parole, per esempio le stop words, compaiono molto frequentemente. Utilizzando un normale algoritmo basato sulla [bag of words](https://en.wikipedia.org/wiki/Bag-of-words_model), i pesi associati a queste parole maschererebbero la vera importanza di altri elementi 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 in cui 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 presenti nel dataset (25000) e df(d,t) è il numero di recensioni che contengono il termine t.

Questi due termini vengono moltiplicati tra loro 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:

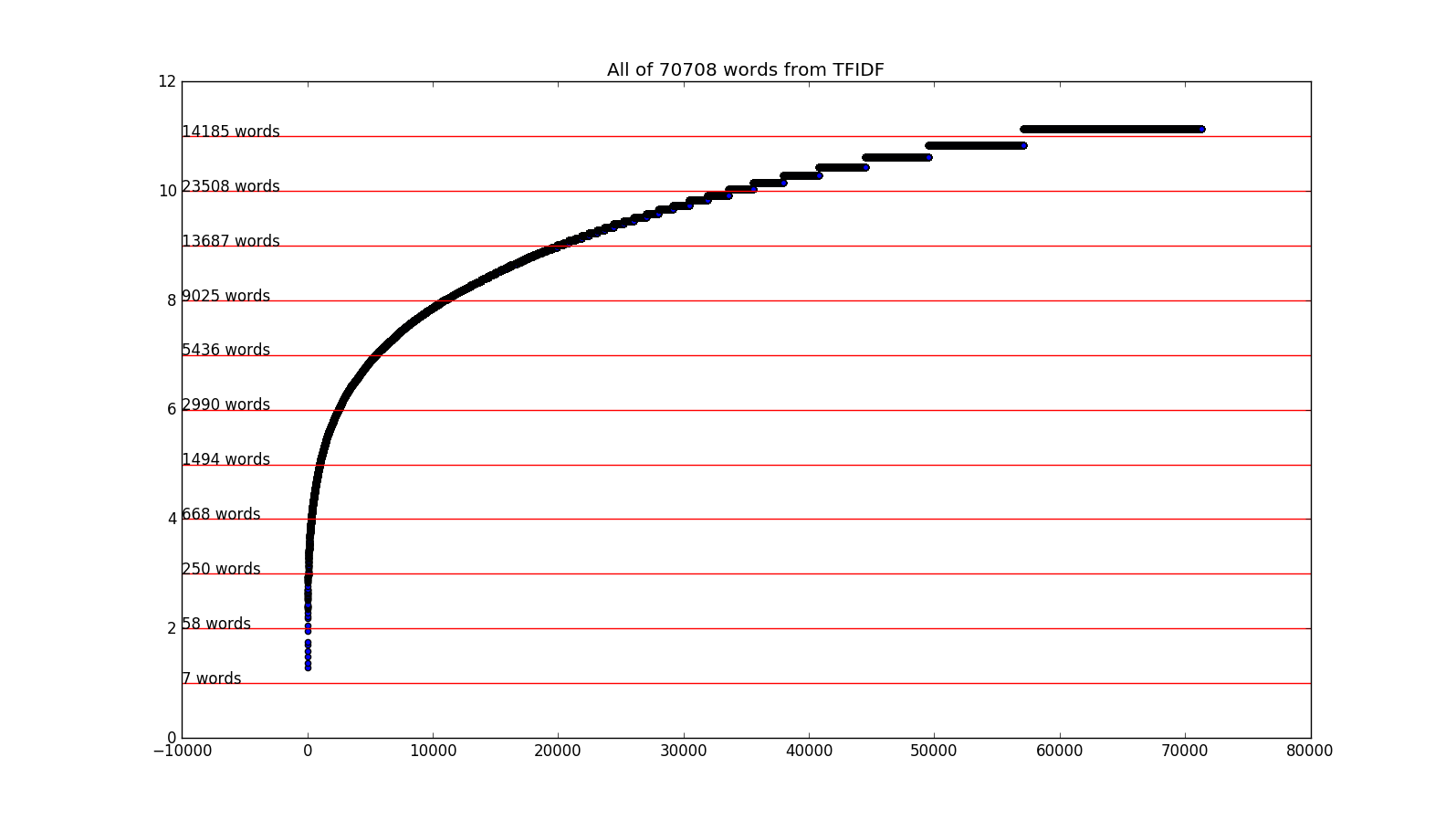
* **Min\_df**: alcune parole hanno una frequenza cosi bassa all’interno della recensione che è possibile ignorarle, per questo ho settato questo parametro a 2, così da eliminare tutte le parole che si ripetono meno di tre volte.
* **Max\_df**: nel caso in cui la nostra precedente pulizia delle stop words avesse tralasciato qualcosa, entrerebbe in gioco questo parametro settato a 0.96. Il suo ruolo è quello di tralasciare le parole che si ripetono per più del 96% all’interno della stessa 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 700000 parole ho deciso di prendere solo le 85000 più rilevanti per evitare di protrarre il tempo di esecuzione oltre i 20 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.
* **Ngram\_range**: il range di valori con cui creare i vari n\_gram, impostato ad (1,3) aggiunge al dizionario elementi contenenti 1, 2 e 3 parole contemporaneamente.

Una volta terminata l’inizializzazione si passa al fittaggio del dataset, la parte in cui la funzione TfidfVectorizer assembla un dizionario di termini e ne assegna i pesi secondo l’algoritmo sopra citato. Qui è stato di fondamentale importanza l’uso dell’Unlabled Train Data che mi ha permesso di aggiungere al dizionario più del 50% dei termini, aumentando l’accuratezza delle predizioni del 3% in media.

Finito il fittaggio sono passato all’effettiva trasformazione del dataset dal domino delle stringhe a quello numerico tramite il metodo “transform” applicato solamente al Train Labled Data e al Test Data. Non avrebbe avuto senso eseguirlo anche sul Unlabled Train Data poichè non avremmo avuto nessun riscontro con cui confrontare una possibile predizione.

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 sono riuscito a plottare il risultato della vettorializzazione:

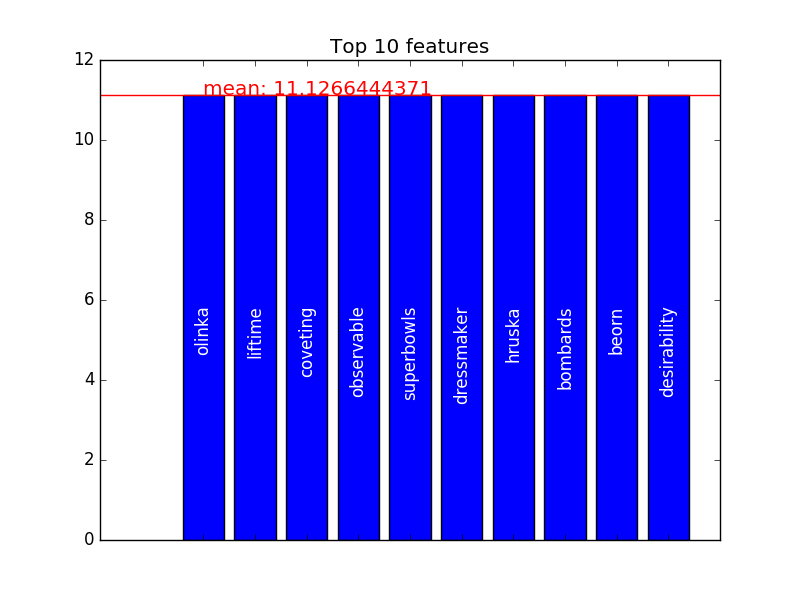


Sull’asse delle ascisse sono riportati il numero delle parole del dizionario creato, mentre sull’asse delle orinate il peso associato. Per una migliore comprensione del grafico ho diviso l’asse y con una serie di rette in rosso, accompagnate da un numero rappresentante le parole che hanno ricevuto un 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 85000.

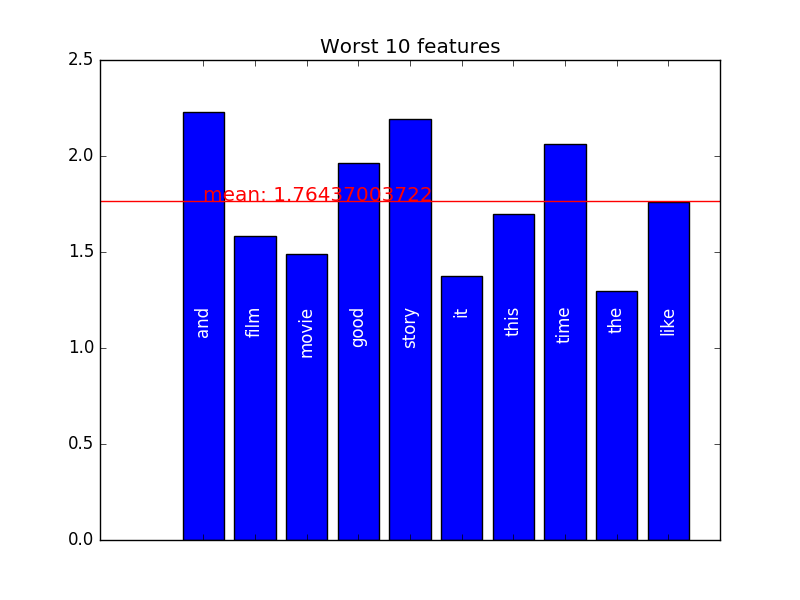
**Dimensionality\_reductionKB**

Dopo aver trasformato le stringhe in vettori, mi sono posto l’obbiettivo di ridurre ulteriormente il numero di features. Il problema con la funzione precedente risiede nel fatto 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 ho plottato le migliori 10 features e le peggiori 20, per il TFIDF, tramite la funzione plot\_top\_n\_words nel file Data\_analisys e ho 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 ho 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 passerò a spiegare la teoria e il meccanismo che sta dietro queste due funzioni.

**Chi2**

Prima di entrare nel particolare ricordiamo a caratteri generali in cosa consiste il chi2.

Il chi2 viene utilizzato per misurare la dipendenza tra variabili stocastiche, in questo caso tra una feature e le Y, cosi facendo eliminiamo le X che risultano indipendenti dalle Y e che quindi non hanno importanza nella predizione.

Ho 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 ho 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 campionamento randomico.
* L’ultimo passaggio sta nel cercare il valore di nelle tabelle del chi2, ricordando di utilizzare il coefficiente 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.

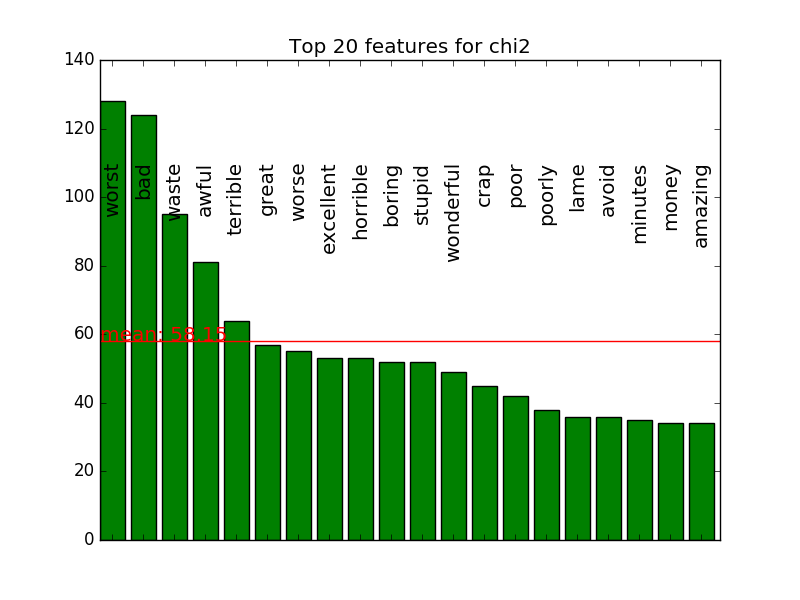
**SelectKBest**

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

* La **score function**: ho usato il chi2 per le ragioni precedentemente citate
* **K**: impostato all’85% di tutte le features presenti nel dataset, rappresenta le miglori caratteristiche da selezionare. Ho deciso di utilizzare 0.85 dopo vari tentatavi e ci siamo accorti che questo era il valore più basso con il quale la differenza nella predizione si manteneva entro limiti accettabili.

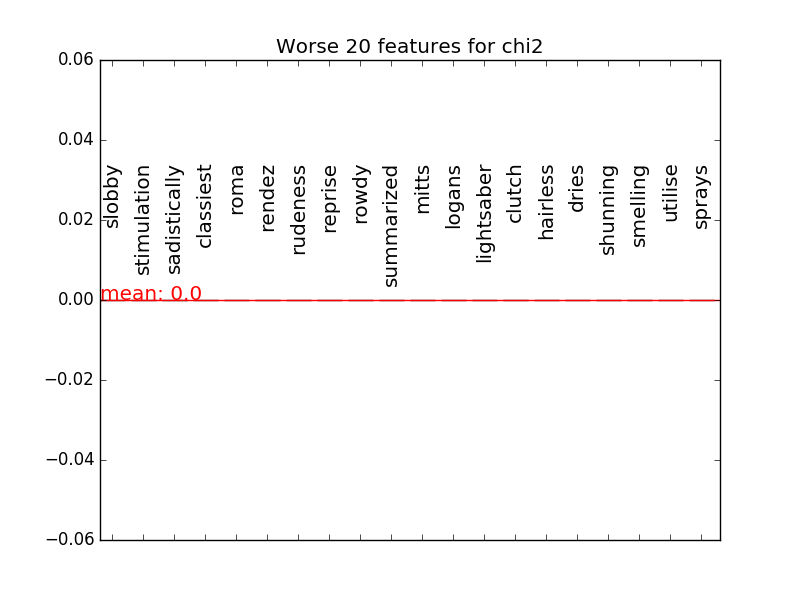
Eseguendo il metodo fit\_transform e passando come paraemtri le X e le Y la funzione esegue (per un tempo medio di 70 secondi) l’algoritmo e ritorna una matrice in cui sono presenti solo l’85% delle migliori features con pesi opportunamente ridimensionati.

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



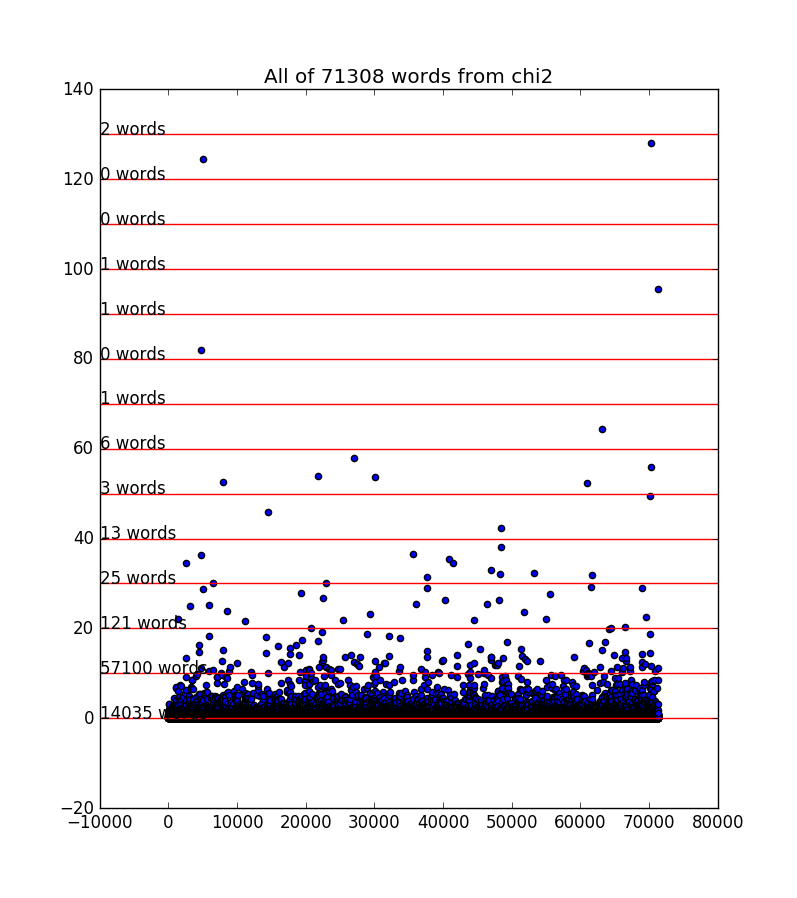
Possiamo notare come la maggior parte delle parole suonino fondamentali per l’analisi dei sentimenti in una recensione. E’ anche interessante vedere come due parole “minutes” e “money”, che per un essere umano possono essere considerate secondarie, siano comunque comprese nella top 20 per l’algoritmo.

Ho anche plottato le peggiori 20 parole:



Anche qui i risultati confermano ciò che potevamo aspettarci.

Plottando tutte 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:



# 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 allenato il 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**

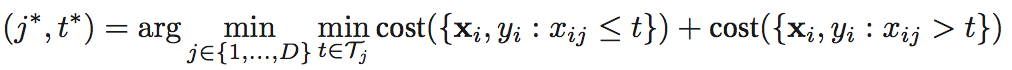
Un albero di decisione funziona in base a più divisioni paralleli lungo l’asse. Ogni divisione indica una regione diversa in cui la feature può trovarsi, queste regioni sono create in base ad una funzione base che andremo a spiegare più avanti e sono affiancate da pesi che specificano la il valore di 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: avendo =2, vengono creati due rami, uno per quando il valore è minore uguale di 2, un altro per quando è strettamente)
* : è la variabile da predire
* : 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 (Maximum Liklehood Estimation) ottimale per la divisione delle migliori features e per la scelta dei 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 Multinoulli, 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, ho optato per la minimizzazione dell’entropia tramite la formula: dove è la MLE per la distribuzione 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 media, non in base al voto sulla stima ma in base alla correlazione che hanno tra loro.

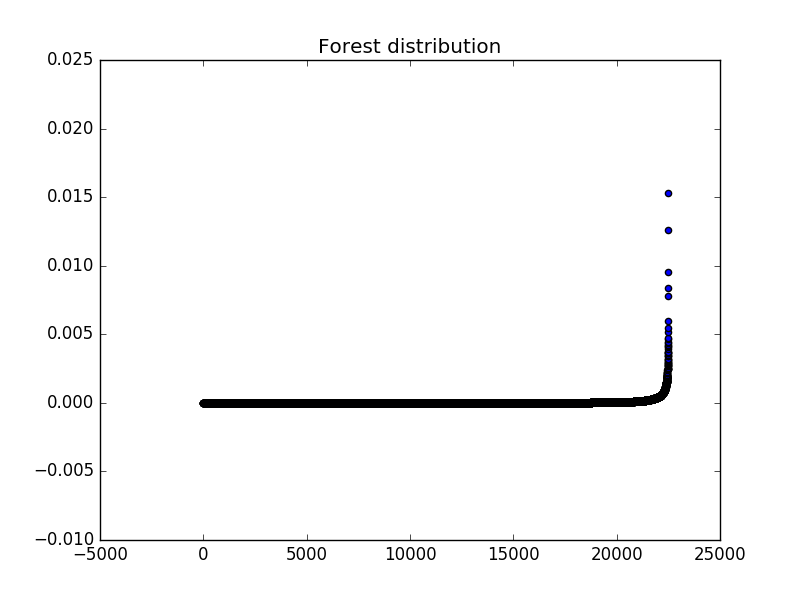
**Parametri adottati**

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

* **n\_estimators**: il numero di alberi della foresta. Ho 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, ho optato per il valore “Entropy” descritto precedentemente.

**Grafici**

In fine ho 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 classificazione.

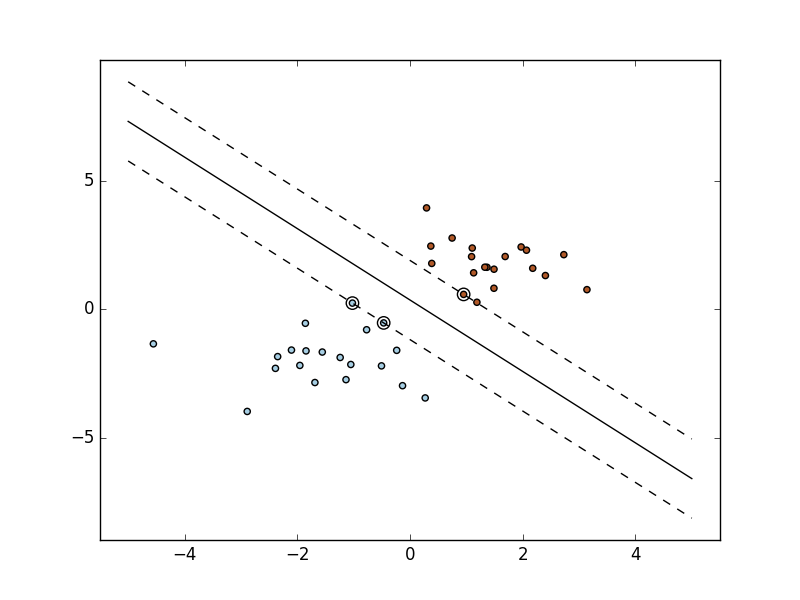
**SVC\_Classifier**

Esattamente come descritto in precedenza, questa funzione fa uso della pulizia del dataset per poi fittare i dataset in un LinearSVC, ovvero un classificatore di tipo Support Vector Machine con kernel lineare.

**Teoria Generale**

Un SVM costruisce una serie di iper-piani in un ampio spazio dimensionale e può essere utilizzato per problemi di regressione o classificazione, come in questo caso. Una buona separazione tra le classi è ottenuta dall’iper-piano che ha la distanza maggiore tra i punti del dataset più ravvicinati tra loro e appartenenti a classi differenti.

Questa distanza è data dalla funzione di margine e ne riporto un esempio qua sotto:



Dobbiamo anche accennare che gli SVM sono una composizione del kernel trick e di una funzione di perdita cosicché la predizione dipenderà solo da un sottoinsieme del dataset chiamato support vectors.

**Kernel**

Come accennato precedentemente il kernel scelto è quello lineare, il che equivale a dire **.**

Ho adottato questo approccio poiché la dimensione delle nostre features è molto alta e, dato che stiamo lavorando con parole, vi è una grande probabilità che la decision boundary sia rappresentabile da una combinazione lienare delle features originali.

L’SVM gode inoltre di due proprietà che li differenzia dagli altri modelli:

* L’uso delle SVM porta sparsità nella funzione di costo piuttosto che nel prior
* Il loro output non è probabilistico

**Kernel Trick**

Il kernel trick ci permette di lavorare con il vettore delle x originali modificando l’algoritmo affinché tutti i prodotti tra x e x’ vengano rimpiazzati con la chiamata al kernel .

**Funzione di costo**

Nelle SVM, in particolare nei problemi di classificazione, la funzione di costo più comunemente applicata è la “Hinge Loss”. Questa consiste nel rimpiazzare la negative log likelihood con in cui è la nostra confidenza nello scegliere y=1.

Introducendo:

* : vettore dei pesi da calcolare
* : variabile di slack
* : upper bound
* : vettori utilizzati per il training
* : numero di support vectors
* : vettore dei pesi iniziale
* : vettore delle labels

La nostra funzione obiettivo sarà:

soggetto a:

La funzione di decisone applicata è la seguente:

in cui