Esercitazione

# Text Mining su Rapporti di Incidenti Aerei

Laurea Magistrale in Ingegneria e Scienze Informatiche
Dip. di Informatica – Scienze e Ingegneria, Cesena

*Insegnamento: Data Mining*

*Modulo di Text Mining*

Prof. Gianluca Moro
gianluca.moro –a-t- unibo.it
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna, sede di Cesena



## Sommario

- Obiettivo dello studio
  - Classificare gli incidenti aerei rispetto al livello di gravità in base a rapporti in linguaggio naturale e variabili strutturate
- Attività dell'esercitazione
  - Generare un primo modello di classificazione basato solamente su attributi strutturati (i.e. nominali, ordinali, intervallo e ratio)
  - Applicando le tecniche di text mining, ripetere l'analisi utilizzando solo gli attributi testuali destrutturati
  - Ripetere il punto precedente utilizzando tutti gli attributi
  - Analizzare la correlazione delle variabili con il grado di gravità dell'incidente (attributo classe)

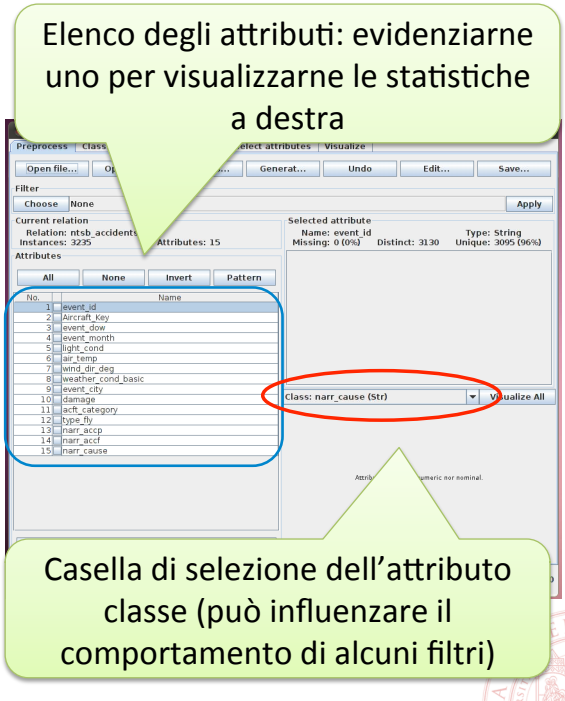
# Data set utilizzato

- Dati su 3.235 **incidenti aeronautici** (2001-03) raccolti dalla *National Transportation Safety Board* (USA)
  - prelevare il file ntsb-causes.arff da http://bit.ly/2hVMg8Q
- Diversi dati riportati per ciascun incidente:
  - identificatore univoco dell'evento (*event_id*)
  - giorno della settimana (*event_dow*) e mese (*event_month*)
  - condizioni atmosferiche (temperatura, luce, vento)
  - classificazione del velivolo
  - classificazione del tipo di volo (osservazione, esibizione, …)
  - descrizione testuale della dinamica e delle conseguenze
  - **classificazione:** entità dei danni al velivolo (4 classi)



# (1) Caricamento dei dati in Weka

- Aprire l'*Explorer* di Weka
- Aprire il file *arff* fornito
- Impostare *damage* come attributo classe
  - spostare per comodità ***damage*** in ultima posizione con il filtro Reorder (unsupervised -> attribute) Reorder 1-7,9-13,**8**
- Osservare gli attributi presenti nei dati, il loro tipo e i valori che possono assumere.



# (2a) Preprocessing: rimozione attributi

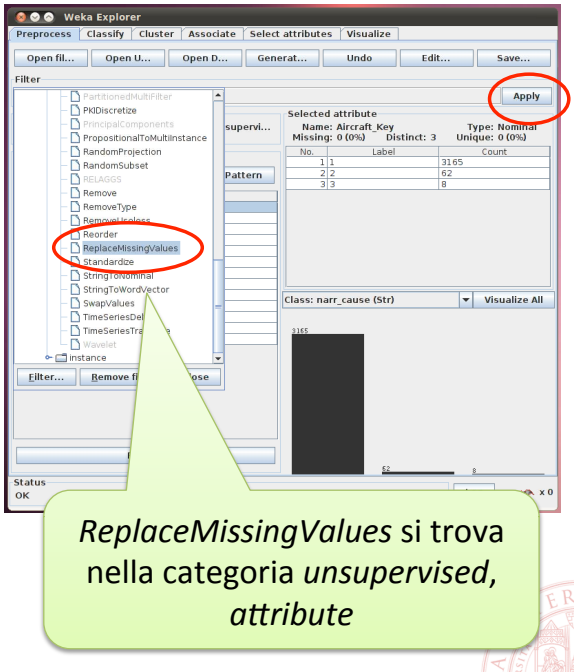
- In generale, attributi con variabilità troppo alta o troppo bassa sono poco utili nella classificazione.
  - L'identificatore degli incidenti ***event_id*** è per definizione unico per ciascuna istanza ed è quindi inutile.
  - 2/3 dei valori dell'attributo ***event_city*** sono distinti, per cui analogamente al precedente può essere rimosso.
  - Allo stesso modo sarebbero scarsamente utili attributi a bassa variabilità, con valori tutti (quasi) uguali tra loro.
- Rimuovere i due attributi indicati selezionandoli (con le caselle di spunta) e premendo *Remove* in basso.
  - In alternativa sarebbe possibile usare il filtro *Remove*

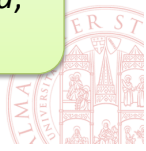




# (2b) Preprocessing: dati mancanti

- Attributi con un numero elevato di valori mancanti (*missing*) possono diventare inutili o inaffidabili.
- Applicare il filtro *ReplaceMissingValues* per sostituire ciascuno di questi con la media o la moda dell'attributo corrispondente.
  - Cliccare su *Choose*, selezionare il filtro dalla lista e cliccare su *Apply*



# (3a) Analisi con attributi strutturati: preprocessing

- Elaboriamo i dati utilizzando per ora solo gli attributi strutturati, quindi eliminiamo gli attributi testuali destrutturati ***narr_accp*, *narr_accf* e *narr_cause***
  - Vedremo successivamente come "nascondere" gli attributi dall'analisi senza eliminarli dal dataset
  - In seguito ripristineremo il dataset in questo stato (prima della rimozione) pertanto salviamolo ora con Save per ricaricarlo dopo (oppure potremmo usare il comando *Undo)*
- Per rimuovere gli attributi, analogamente a prima, selezionarli e cliccare su *Remove*

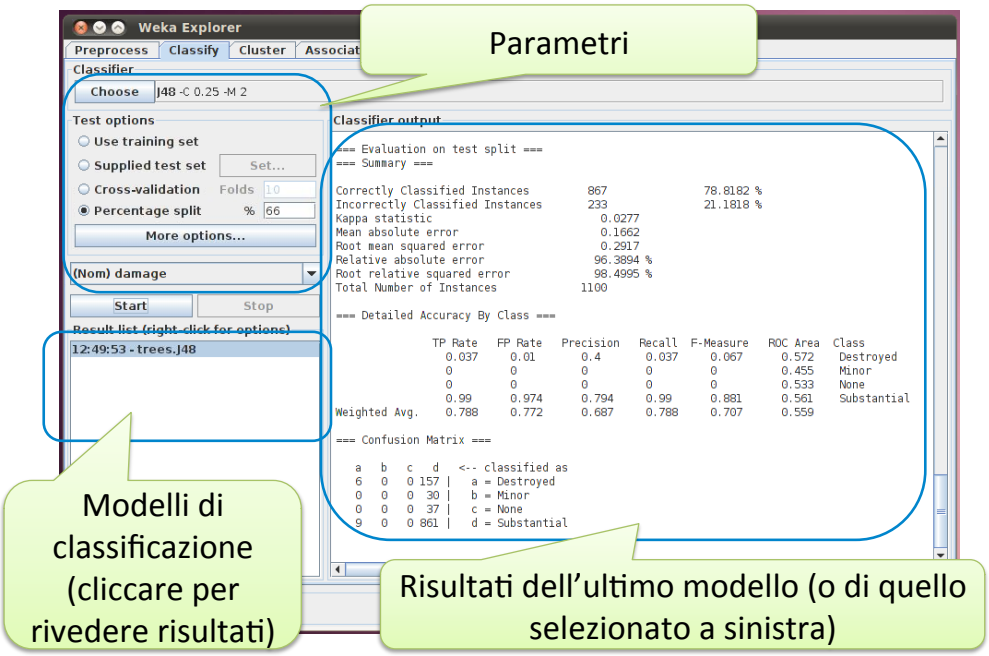




# (3b) Analisi con attributi strutturati: classificazione (i)

- Passare alla scheda ***Classify*** dell'*Explorer* per eseguire gli algoritmi di classificazione sui dati
- Cliccare su *Choose* e selezionare l'algoritmo di classificazione ***J48*** nella categoria ***trees*** (lasciare i parametri di default)
- Selezionare nel riquadro ***Test options*** la modalità di training/ test ***Percentage split*** e lasciare "66" nel campo "%"
  - In questo modo, 2/3 delle istanze del dataset (il 66%) sono usate come training set, il restante 1/3 come test set
- Selezionare *Damage* come attributo classe (l'attributo da predire in base agli altri)
- Cliccare su *Start* per generare e testare il modello di classificazione

# (3b) Analisi con attributi strutturati: classificazione (ii)








# (3b) Analisi con attributi strutturati: modalità di training e test

Effettuiamo gli stessi esperimenti cambiando il metodo di valutazione (*Test options*).

- Con ***Use training test***, viene usato l'intero dataset sia per l'addestramento del modello che per la sua valutazione: questo di norma produce risultati migliori, ma meno attendibili (training e test set dovrebbero essere disgiunti).
- Con ***Cross validation***, il dataset viene diviso equamente in *N* parti (o *fold*, 10 di default), ciascuna delle quali viene usata per testare un modello addestrato sui dati delle restanti *N-1* parti; alla fine vengono mostrati i risultati congiunti degli *N* test effettuati.
- (vedremo ***Supplied test set*** successivamente...)

# (3b) Analisi con attributi strutturati: interpretazione dei risultati (i)

- *Correctly Classified Instances* mostra l'**accuratezza**, intesa come numero e percentuale rispetto al totale di istanze di test per cui è prevista la classe corretta
- Per ogni classe, ***Precision* e *Recall*** indicano i veri positivi di una classe in rapporto rispettivamente ai positivi predetti e a quelli reali. ***F-Measure*** è la media armonica di questi due valori
- In fondo ai risultati, la **matrice di confusione** mostra dettagliatamente il numero di istanze di test per ciascuna classe reale (righe) e predetta (colonne)





# (3b) Analisi con attributi strutturati: Matrice di Confusione

| | | classe predetta | | | |
|---|---|---|---|---|---|
| | | destroyed | minor | none | substantial |
| classe reale | destroyed | 6 | 0 | 0 | 157 |
| | minor | 0 | 0 | 0 | 30 |
| | none | 0 | 0 | 0 | 37 |
| | substantial | 9 | 0 | 0 | 861 |

$$\text{recall}(x) = R_x = c_{x,x} \Big/ \sum_i c_{x,i}$$

$$\text{precision}(x) = P_x = c_{x,x} \Big/ \sum_i c_{i,x}$$

$$\text{F-measure}(x) = 2R_xP_x \Big/ R_x + P_x$$

| classe | rec. | prec. | F-m. |
|---|---|---|---|
| destr. | 0,04 | 0,4 | 0,07 |
| minor | 0 | 0 | 0 |
| none | 0 | 0 | 0 |
| subst. | 0,99 | 0,79 | 0,88 |

$c_{i,j}$ = numero di istanze di test di classe *i* per cui il modello ha predetto la classe *j*

La somma di tutti i valori è il numero di istanze di test classificate. La somma di quelli sulla diagonale principale è il num. di istanze classificate correttamente per ogni classe

# (3b) Analisi con attributi strutturati: interpretazione dei risultati (ii)

- Nell'esempio, circa il 79% delle istanze di test è classificato correttamente, ma ....
- ... dalla matrice di confusione, si nota che l'algoritmo ha classificato come damage="Substantial" quasi tutte le istanze di test (tutte tranne 15 su 1.100, classificate in "Destroyed")
- L'accuratezza è relativamente alta solo perché il 79,6% delle istanze è di questa classe
- **Recall e precision delle altre classi sono nulle o molto basse** perciò il modello di classificazione generato è molto scarso
- Lo sbilanciamento tra le classi tipicamente rende più difficile la generazione di un buon modello: alcuni algoritmi gestiscono meglio di altri questo problema





# (3c) Analisi con attributi strutturati: generazione di un nuovo modello

- Usiamo un altro decision tree sugli stessi dati...
- Cliccare su *Choose* e selezionare l'algoritmo *RandomForest* (sempre nella categoria *trees*)
- Lasciare le altre opzioni come prima
  - Modalità di test *Percentage split*, 66%; classe *damage*
- Avviare l'algoritmo con *Start*
- Eseguire un'ulteriore test con uguali opzioni usando l'algoritmo *SMO* (categoria *function*)
- Analizzare i nuovi risultati e confrontarli con i precedenti...

# (3c) Analisi con attributi strutturati: confronto tra modelli

- L'accuratezza con RandomForest è circa 76%, inferiore a prima
- Tuttavia, precision e recall delle classi meno rappresentate ora sono superiori (seppure ancora basse)
- Il modello distingue meglio del precedente le istanze delle classi con meno istanze

```
=== Detailed Accuracy By Class ===

               Precision  Recall  F-Measure  Class
               0.259      0.135   0.177      Destroyed
               0.333      0.200   0.250      Minor
               0.538      0.189   0.280      None
               0.810      0.916   0.860      Substantial
Weighted Avg.  0.706      0.756   0.723

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
  22   0   2 139 |   a = Destroyed
   2   6   0  22 |   b = Minor
   2   2   7  26 |   c = None
  59  10   4 797 |   d = Substantial
```

- *Utilizzare SVM eseguendo l'algoritmo SMO e confrontare i risultati con quelli di J48 e RandomForest*



# (4) Analisi con soli attributi testuali destrutturati – ***text mining***

- Applichiamo le tecniche di text mining per sfruttare i dati testuali destrutturati a disposizione
- Ripetiamo l'analisi utilizzando solo gli attributi destrutturati
- Ritornare alla scheda *Preprocess* e recuperare il dataset con gli attributi *narr_accp*, *narr_accf* e *narr_cause*
  - ricaricare il file salvato prima oppure usare il comando *Undo*, assicurandosi che siano rimasti applicati i passaggi di pre-processing dei punti 2a e 2b

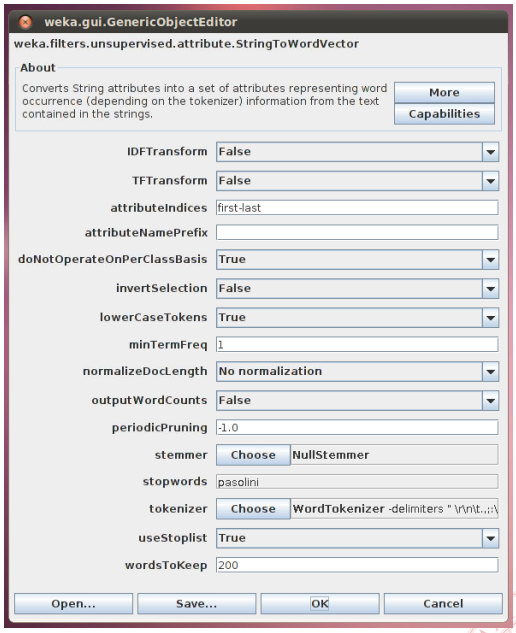
# (4a) Analisi con attributi destrutturati: estrazione delle feature dal testo (i)

- Per fare text mining occorre strutturare gli attributi destrutturati
- Il filtro *StringToWordVector* di Weka esegue i passaggi tipici previsti per questa trasformazione:
  - Estrazione delle singole parole dai testi (*tokenization*)
  - Rimozione e/o trasformazione (*stemming*) delle parole per ridurne il numero
  - Selezione delle parole più rilevanti (*TF, TF-IDF ...*)
  - Aggiunta di nuovi attributi che rappresentano la presenza delle parole selezionate (in sostituzione di quelli originali)



# (4a) Analisi con attributi destrutturati: estrazione delle feature dal testo (ii)

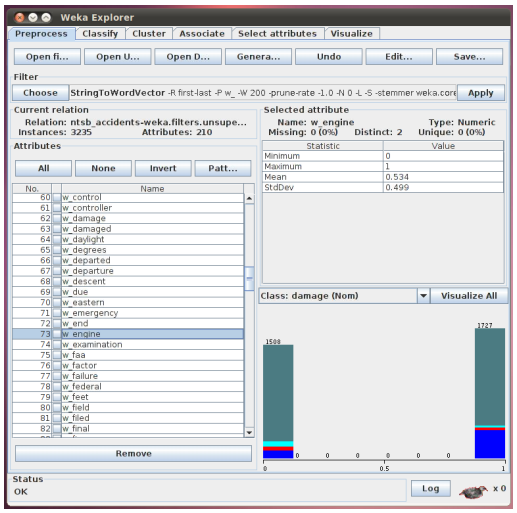
- Selezionare il filtro *StringToWordVector (unsupervised, attribute)* ed impostarne le opzioni (lasciare invariate le altre):
  - *attributeNamePrefix* = "w_"
  - *doNotOperateOnPer ClassBasis* = true
  - *lowerCaseTokens* = true
  - *useStopList* = true
  - *wordsToKeep* = 100




# (4a) Analisi con attributi destrutturati: estrazione delle feature dal testo (iii)

Applicando il filtro (*Apply*), il dataset si trasforma

- Gli attributi destrutturati (string) sono rimossi
- Sono creati 100 attributi "w_*parola*", che valgono 1 nelle istanze dove *parola* era presente, 0 altrove: **rappresentazione VSM con frequenza binaria**








# (4b) Analisi solo con attributi destrutturati: generazione del modello

- Generiamo il modello di classificazione usando <u>solamente</u> le feature/parole estratte dal testo:
  - Salvare i dati nello stato corrente in un nuovo file (in alternativa, si potrà ripristinare il dataset con *Undo*)
  - Rimuovere tutti gli attributi usati nell'analisi precedente, lasciare solamente *damage* e gli attributi "w_"
  - Ritornare alla scheda *Classify*, selezionare le opzioni usate nel primo esperimento e generare il modello (*start*):
    - algoritmo: *J48*,
    - opzione di test: Percentage split ,66%
    - classe *damage*

# (4b) Analisi solo con attributi destrutturati: misure di efficacia del modello

- Il tempo impiegato è aumentato (analisi su più attributi) di pochi secondi
- L'accuratezza è però migliorata di molto (quasi il 90%).
- Anche recall e precision delle singole classi sono migliorate.

```
Time taken to build model: 2.97 seconds

=== Evaluation on test split ===

Time taken to test model on training split: 0.02 seconds

=== Summary ===

Correctly Classified Instances         985               89.5455 %
Incorrectly Classified Instances       115               10.4545 %
Kappa statistic                          0.6776
Mean absolute error                      0.0695
Root mean squared error                  0.224
Relative absolute error                 40.3151 %
Root relative squared error             75.6299 %
Coverage of cases (0.95 level)          92        %
Mean rel. region size (0.95 level)      37.3409 %
Total Number of Instances             1100

=== Detailed Accuracy By Class ===

               Precision  Recall  F-Measure  Class
                 0.821     0.761   0.790      Destroyed
                 0.440     0.367   0.400      Minor
                 0.357     0.135   0.196      None
                 0.929     0.971   0.949      Substantial
Weighted Avg.    0.880     0.895   0.885

=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 124   1   6  32 |   a = Destroyed
   1  11   3  15 |   b = Minor
   6   8   5  18 |   c = None
  20   5   0 845 |   d = Substantial
```





# (4b) Analisi solo con attributi destrutturati: generazione di un nuovo modello

- Eseguire un ulteriore test con l'algoritmo *DMNBText* (categoria *bayes*) lasciando le opzioni di default.
  - DMNBText è una variante del *NaiveBayes* specializzata per vettori di parole (come quelli ricavati con *StringToWordVector*).
  - In questo caso, si ottiene un risultato appena inferiore a quello precedente (accuratezza circa 88%) con un tempo di esecuzione nettamente inferiore (quasi istantaneo)
  - I livelli di precision e recall per le varie classi subiscono alcuni cambiamenti di rilievo tra un caso e l'altro

# (4c) Analisi con attributi strutturati e destrutturati

- Ritornare alla scheda *Preprocess* e ripristinare gli attributi rimossi all'inizio del passaggio precedente.
  - se è stato salvato un file ricaricarlo, altrimenti usare *Undo*
- Ritornare a *Classify*, selezionare nuovamente le opzioni "base" (*J48*, *Percentage split* 66%, classe *damage*) e avviare la classificazione.





# (4c) Analisi con attributi strutturati e destrutturati: classificazione

- Con J48, il risultato ottenuto con tutti gli attributi è leggermente superiore a quello ottenuto con le sole feature testuali.
  - La tabella a destra riassume i risultati ottenuti negli esperimenti proposti (con *perc. split 66%*).

S = solo attributi "strutturati"
T = solo attributi ricavati dal testo
S+T = tutti gli attributi

| Algoritmo | M. | S | T | S+T |
|---|---|---|---|---|
| J48 | acc. | 78,8% | 89,5% | 90,2% |
| | F1 | 70,7% | 88,5% | 89,2% |
| Random Forest | acc. | 75,6% | 87,0% | 84,7% |
| | F1 | 72,3% | 85,3% | 82,4% |
| DMNBText | acc. | - | 88,1% | - |
| | F1 | - | 86,4% | - |

Misure
- accuratezza (istanze corrette)
- media pesata della misura F1

# Confronto di Recall e Precision con gli esperimenti precedenti

| Classe (# ist. test) | Misura | J48 S | RndFor S | J48 T | RndFor T | DMNB T | J48 S+T | RndFor S+T |
|---|---|---|---|---|---|---|---|---|
| destroyed (163) | recall | 0.04 | 0.14 | 0.76 | 0.65 | 0.71 | 0.81 | 0.49 |
| | prec. | 0.40 | 0.26 | 0.82 | 0.82 | 0.80 | 0.81 | 0.79 |
| minor (30) | recall | 0.00 | 0.20 | 0.37 | 0.10 | 0.07 | 0.33 | 0.17 |
| | prec. | 0.00 | 0.33 | 0.44 | 0.25 | 1.00 | 0.42 | 0.39 |
| none (37) | recall | 0.00 | 0.19 | 0.14 | 0.16 | 0.22 | 0.16 | 0.11 |
| | prec. | 0.00 | 0.54 | 0.36 | 0.43 | 0.44 | 0.50 | 0.40 |
| substantial (870) | recall | 0.99 | 0.92 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 |
| | prec. | 0.79 | 0.81 | 0.93 | 0.89 | 0.90 | 0.94 | 0.86 |
| *media pesata* | recall | 0.79 | 0.76 | 0.90 | 0.87 | 0.88 | 0.90 | 0.85 |
| | prec. | 0.69 | 0.71 | 0.88 | 0.85 | 0.87 | 0.89 | 0.82 |

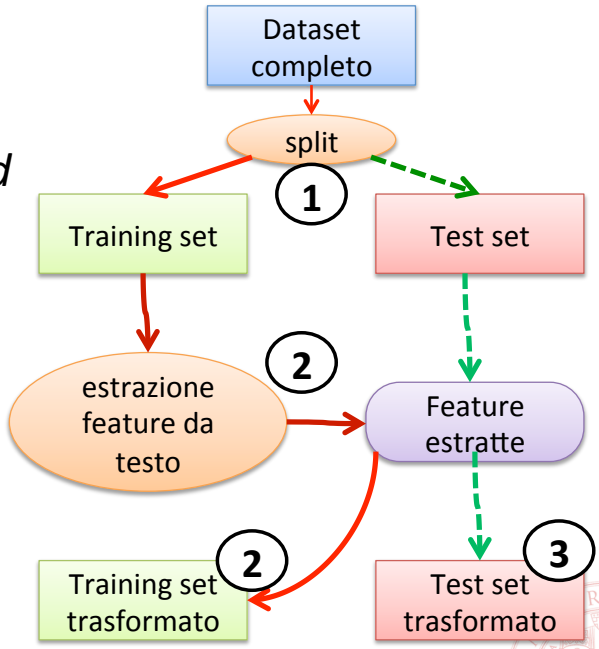


# WEKA: applicare trasformazioni al dataset compatibili con uno scenario a regime (i)

- Negli esempi precedenti il filtro *StringToWordVector* è stato applicato all'intero dataset, che in seguito è stato diviso in training e test set prima della classificazione
- In questo modo, *StringToWordVector* ha selezionato parole (feature) basandosi anche sui dati usati per testare il modello
- A regime di un caso reale, documenti e dati da classificare sono di solito ignoti nella fase di training, infatti ogni trasformazione applicata al training set è inferita solo sulla base dei dati presenti nel training set stesso
- Per classificare nuovi dati/documenti, le stesse trasformazioni si applicano ad ogni documento da classificare per renderli compatibili con il modello

# WEKA: applicare trasformazioni al dataset compatibili con uno scenario a regime (ii)

1) Dividere il dataset in training e test set
2) Applicare *StringToWord Vector* al solo training set per estrarre le feature
3) Trasformare il test set affinché abbia lo stesso insieme di feature definito al punto 2






# *FilteredClassifier*

- Per applicare in WEKA il processo descritto in precedenza, si può utilizzare *FilteredClassifier*
  - *meta-algoritmo* che (i) applica un filtro scelto dall'utente ai dati di training, (ii) genera un modello di classificazione dal training set trasformato, (iii) applica il risultato del filtro di trasformazione al test set, (iv) classifica il test set.
- Con *StringToWordVector* in *FilteredClassifier*
  - prima il training set è usato per estrarre le feature e trasformato di conseguenza,
  - poi il test set è trasformato riutilizzando il filtro con le stesse feature.

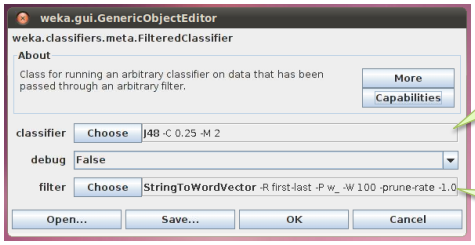
# (5a) Analisi scenario a regime: tutti gli attributi (i)

- Riprendiamo l'esercizio: copiare negli appunti la configurazione usata per *StringToWordVector*
  - clic destro su casella dei parametri → *Copy configuration to clipboard*
- Riportare il dataset allo stato in cui sono presenti gli attributi "*narr_*" di tipo stringa (ricaricare un file salvato o usare *Undo*)
- Passare alla scheda *Classify*, selezionare come algoritmo *FilteredClassifier* (categoria *meta*) ed aprirne le impostazioni





# (5a) Analisi scenario a regime: tutti gli attributi (ii)




Algoritmo di classificazione incapsulato, da utilizzare sulle istanze filtrate

Filtro da applicare alle istanze

- Lasciare *J48* come classificatore e impostare come filtro *StringToWordVector* con i parametri usati prima
  - per incollare la configurazione copiata: clic destro sulla casella col filtro → *Enter configuration...* → incollare la configurazione nella casella di testo che appare (Ctrl+V) → *OK*
- Confermare i parametri, impostare *Percentage split* a 66% e classe *damage* e avviare la classificazione

# (5b) Analisi scenario a regime: solo con attributi destrutturati

- Per compiere l'analisi sui soli attributi testuali destrutturati utilizzando questa procedura, è sufficiente rimuovere gli altri attributi:
  - Rimuovere dall'Explorer tutti gli attributi da non utilizzare (lasciare solo *damage* e gli attributi "narr_...")
  - Vedremo poi che in alternativa si può utilizzare in *FilteredClassifier* il filtro *MultiFilter* per combinare insieme i filtri *Remove* e *StringToWordVector*
- Usare nuovamente *FilteredClassifier* con il filtro *StringToWordVector* impostato prima per classificare i dati con *J48* e *DMNBText*





# (5b) Analisi scenario a regime: risultati

- I nuovi risultati (ultime due colonne) sono simili a quelli di prima (alcuni di poco migliori, altri di poco peggiori).
- Questi risultati sono però più attendibili: rispecchiano meglio scenari d'uso reali.
- Con *FilteredClassifier* sarà inoltre più semplice testare diversi parametri di *StringToWordVector*...

S = solo attributi "strutturati"
T = solo attributi ricavati dal testo
S+T = tutti gli attributi

| Algoritmo | M. | S | T | S+T |
|---|---|---|---|---|
| J48 | acc. | 78,8% | 89,8% | 90,3% |
| | F1 | 70,7% | 88,9% | 89,1% |
| Random Forest | acc. | 75,6% | 87,1% | 86,3% |
| | F1 | 72,3% | 85,3% | 84,1% |
| DMNBText | acc. | - | 89,2% | - |
| | F1 | - | 87,4% | - |

Misure
- accuratezza (istanze corrette)
- media pesata della misura F1

# (6) Variazione dei parametri di *StringToWordVector*

Nelle prossime slide analizziamo gli effetti della variazione di alcuni parametri del filtro *StringToWordVector*, con cui si modifica l'insieme di attributi (i.e. feature) ricavati dal testo

- Riportare il dataset allo stato in cui sono presenti tutti gli attributi strutturati e non, tranne i due "event_" rimossi all'inizio (ricaricare un file o *Undo*)
- Nella scheda *Classify*, lasciare selezionato *FilteredClassifier* e dalla sua finestra dei parametri aprire quella di *StringToWordVector*...



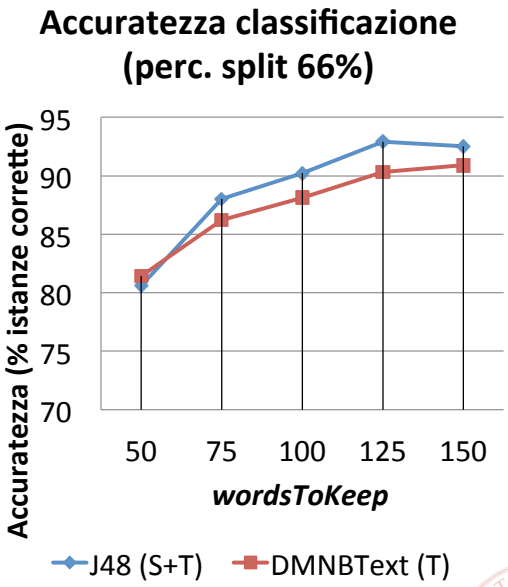


# (6a) *StringToWordVector*: numero di parole

- *StringToWordVector* raccoglie tutte le parole distinte dai documenti e seleziona quelle che compaiono più frequentemente nelle istanze.
- Il parametro *wordsToKeep* stabilisce quante parole mantenere
- Impostare *wordsToKeep* a 50 mantenendo i valori usati prima per gli altri parametri, selezionare *J48* come classificatore ed avviare la classificazione
- Ripetere gli stessi passaggi con *wordsToKeep* = 150

# (6a) *StringToWordVector*: numero di parole

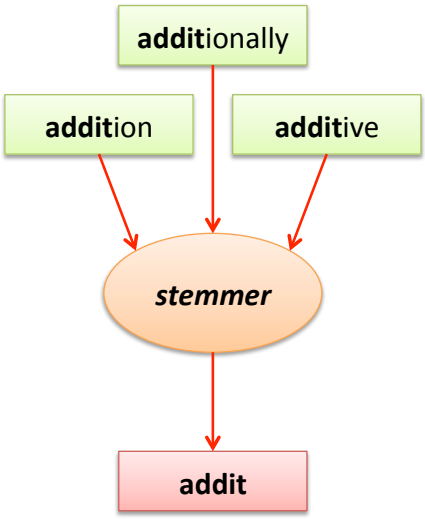
- In generale, un numero minore di feature produce un risultato peggiore, mentre aumentandole i risultati migliorano
- D'altra parte, le nuove feature oltre una data soglia sono progressivamente meno utili e non migliorano i risultati
  - Ad esempio, nella figura, i risultati di J48 peggiorano dopo 125 feature






# Stemming

- Gli algoritmi di ***stemming*** sono usati per ridurre ogni parola ad una forma radice (non necessariamente la radice morfologica)
- **Parole distinte con la stessa radice**, di solito correlate anche semanticamente, **generano una sola feature**
- La radice risultante in generale non è una parola lessicalmente corretta (come nell'esempio a destra)



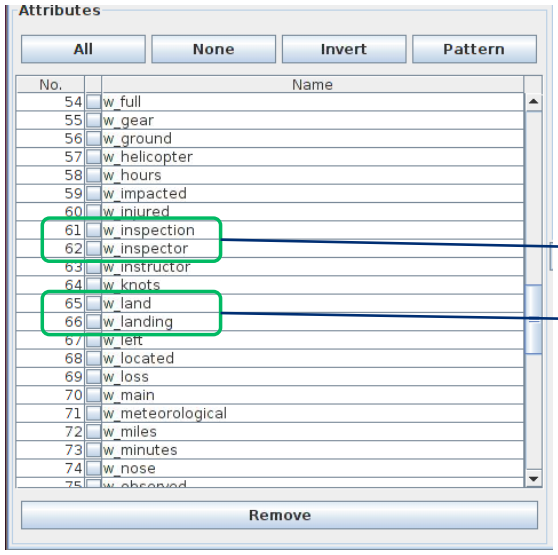
# (6b) *StringToWordVector*: stemming

- In *Preprocess*, applicare *StringToWordVector* impostando *stemmer* a *IteratedLovinsStemmer*.
- Osservare le feature generate: appaiono diverse parole “troncate” per via dello stemming.
- Usare *Undo* per ripristinare gli attributi stringa.
- In Classify, impostare come sopra lo stemming dentro *FilteredClassifier* e rieseguire la classificazione (con *wordsToKeep* = 100). A parità di numero di feature, si nota un aumento di accuratezza.
- Dopo questo test, reimpostare lo stemming a *Null*

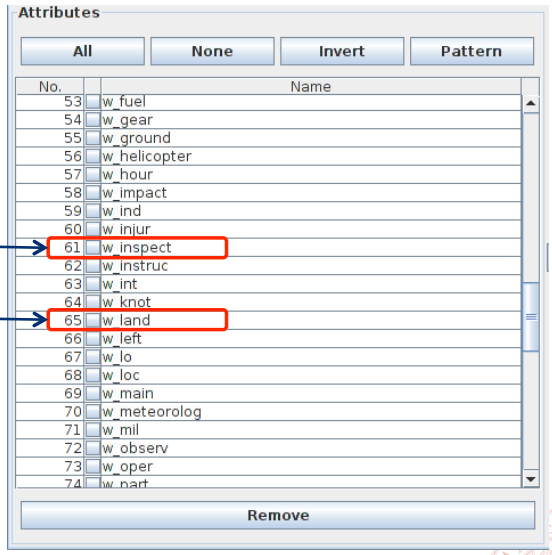


# Confronto tra feature generate con e senza stemming

***NullStemmer* (no stemming)**




***IteratedLovinsStemmer***



# (6c) *StringToWordVector*: estrazione feature su classi separate

- Finora abbiamo estratto e selezionato le parole da tutti i dati di training presi come un unico insieme.
- Tuttavia, conoscendo le classi dei dati di training, è possibile estrarre un insieme di parole per ciascuna classe e alla fine considerare l'unione degli insiemi.
- Così si garantisce che anche le classi meno popolate siano ben rappresentate dalle feature selezionate.
- Impostando l'opzione ***doNotOperateOnClassBasis*** di *StringToWordVector* a *false* (default), sono estratte almeno *wordsToKeep* parole per ciascuna classe.



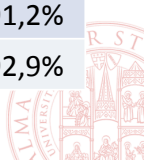


# (6c) *StringToWordVector*: estrazione feature su classi separate

- Rieseguire l'esperimento usando l'algoritmo *J48* su tutti gli attributi e con *doNotOperateOnClassBasis* = false e *wordsToKeep* = 100.
  - Vengono generate molto meno di 4x100=400 parole, perché molte si ripetono in più classi.
- Eseguire di nuovo con *wordsToKeep* a 25 e 50.
  - Con 50, si ha un risultato di poco migliore con un numero di feature di poco più basso.

**CB** = selezione basata su classi (DNOOCB=false) o no (DNOOCB=true)
**WtK** = parametro *wordsToKeep*
**feature** = numero feature risultanti dai testi
**acc.** = accuratezza con *FilteredClassifier*, *J48*

| CB | WtK | feature | acc. |
|---|---|---|---|
| no | 100 | 100 | 90,3% |
| sì | 25 | 47 | 83,2% |
| sì | 50 | 96 | 91,2% |
| sì | 100 | 190 | 92,9% |



# Altre opzioni di *StringToWordVector*

- Il filtro dapprima scompone ogni testo in singole parole e compie alcune trasformazioni e selezioni
  - *tokenizer* è l'algoritmo di divisione del testo in parole
  - *lowerCaseTokens* indica se uniformare le parole a lettere minuscole
  - *useStopList* indica se rimuovere le stopword, parole solitamente non utili ai fini dell'analisi
- In seguito viene creata una lista di parole distinte trovate nei documenti e per ogni coppia parola-documento viene calcolato un valore da riportare nel dataset risultante
  - *outputWordCounts*: se true si prende il numero di occorrenze delle parole nei testi, altrimenti si considera solo la presenza (1) o assenza (0)
  - *TFTransform*: converte ogni valore *x* calcolato in log(1+*x*), in genere si usa in combinazione con *outputWordCounts*
  - *IDFTransform*: moltiplica ogni valore per il fattore IDF, calcolato come log( {totale istanze} / {istanze in cui compare la parola} )





# Selezione attributi in Weka

- Dalla scheda *Select attributes* è si esegue la selezione di un sottoinsieme di attributi più informativi
  - gli attributi selezionati sono solo mostrati: per modificare effettivamente il dataset, usare il filtro *AttributeSelector*
- La selezione prevede 2 operazioni (per entrambi sono disponibili diversi algoritmi):
  - Il *metodo di valutazione* (*Attribute Evaluator*) indica come è valutato ciascun singolo attributo o insieme di essi.
  - Il *metodo di ricerca* (*Search Method*) indica quali attributi o insiemi valutare e quando interrompere la ricerca (valutare tutti gli insiemi possibili richiederebbe troppo tempo).

# (7) Selezione dei termini più significativi

- In *Preprocess*, applicare (ai dati con gli attributi stringa) il filtro *StringToWordVector* con i parametri usati all'inizio (*wordsToKeep* = 100, no stemming).
- Nella scheda *Select attributes*, lasciare gli algoritmi selezionati di default (*CfsSubsetEval* e *BestFirst*), selezionare *damage* come classe nella casella sotto e cliccare su *Start* per avviare la selezione.
- Osservare l'elenco di attributi selezionati: questi corrispondono alle parole presenti nei testi che più di altre aiutano a comprendere l'entità dei danni negli incidenti
- Ripetere la prova con il metodo di selezione *ChisquaredAttributeEval*





# **Esercizio:** spam filtering

- Il dataset *mailspam* contiene i **testi di 2.893 e-mail**, ciascuna classificata come ***spam* o *non-spam***
  - prelevare il file da http://bit.ly/2zJhQBW
- Vogliamo prevedere la classe delle mail dal testo
  - Questo dataset risulta più facile da trattare del precedente poiché ha solo 2 classi ben distinte tra loro
- Negli esperimenti di classificazione, utilizzare i parametri specificati in precedenza:
  - *FilteredClassifier* con filtro *StringToWordVector* con opzioni *doNotOperateOnClassBasis*, *lowerCaseTokens* e *useStoplist* attive e classificatore alternato tra *J48* e *DMNBText*
  - Modalità di test *Percentage split* con perc. training 66%

# **Esercizio:** spam filtering – Quesiti

- Fare un primo test con *wordsToKeep*=50, quale algoritmo di classificazione tra *J48* e *DMNBText* restituisce un modello migliore?
  - In entrambi i casi l'accuratezza dovrebbe essere > 95%
- In entrambi i casi, quale delle due classi ha Precision e Recall inferiori ?
- Per ciascun algoritmo, indicativamente, per quale valore minimo di *wordsToKeep,* precision e recall della medesima classe superano entrambe il 90% ?