Progetto Analisi dei dati

Francesca Rizzo

23 febbraio 2021

Indice

Introduzione

Lo scopo del progetto è implementare un predittore in grado di predire il codice MSC di un articolo dati titolo e abstract. Il codice usato, e qualche commento ulteriore sull'implementazione, si trova su Colab al seguente link.

I dati di training e validation forniti sono composti da due stringhe (titolo e abstract dell'articolo) e una lista di codici MSC associati ad ogni articolo. Ho scelto di utilizzare come output dei dati di training solo il primo MSC code, che ho registrato con attributo `msc1`. Per valutare la precisione dei metodi usati ho dichiarato corretta una classificazione in cui l'MSC code predetto appartiene alla lista degli MSC code forniti.
Tra i vari metodi utilizzati quello con risultato migliore è stato **Naive Bayes**, sul quale tramite validazione ho ottenuto una precisione del **55.2**%.

In questa relazione presento le scelte fatte per convertire i dati testuali in dati numerici ai quali poter applicare i metodi di classificazione studiati ([1]); i modelli testati ([2]) e il modello migliore trovato ([3]); e infine alcuni dei miglioramenti che potrebbero essere fatti per ottenere risultati più soddisfacenti ([4]).

1 Vettorizzazione dell'input

Per "vettorizzare" i dati di input ho creato un vocabolario V a partire dalle parole presenti nel training set: in questo modo ad ogni testo t è possibile associare il vettore $v_t = (\varphi(l,t))_{l \in V}$, dove $\varphi(l,t)$ è il numero di volte che la stringa l appare nel testo t.
A partire dal testo costruiamo V con alcuni degli n-grammi (segmenti di n parole consecutive) presenti nei testi dei dati di input. Ovviamente per costruire un vocabolario che sia utile per la classificazione dobbiamo scegliere fino a quale n considerare gli n-grammi (nel modello migliore n sarà uguale a 2), e anche in base a che criteri includere (o meno) un n-gramma presente in un testo: ad esempio sequenze tipo *"this article"* saranno presenti in tutti gli articoli, e quindi non utili alla classificazione.
Per praticità nel seguito indicheremo con "termine" sia le parole singole che gli n-grammi.

Per gestire la creazione del vocabolario e la vettorizzazione dei testi ho usato il pacchetto `text2vec`. La funzione `itoken` prende in input un vettore di testi e restituisce un iteratore che scorre i vari "token" del testo, nel nostro caso le parole (rimuovendo spazi e segni di punteggiatura). La funzione `create_vocabulary` prende in input un iteratore su una stringa e un vettore (a, b) e restituisce gli n-grammi con $a \leq n \leq b$ presenti, con delle statistiche (numero di volte che compare il termine e numero di documenti in cui è presente).

```
it_train = itoken(_text_, tokenizer = word_tokenizer)
vocab = create_vocabulary(it_train, ngrams = c(1,n))
pruned_vocab = prune_vocabulary(vocab, term_count_min = term_min,
                                doc_proportion_max = doc_prop_max,
                                doc_proportion_min = doc_prop_min)
```

Il vocabolario così creato ha moltissime parole (circa 10^5), tante delle quali poco significative perché presenti troppo poche volte (come ad esempio i numeri delle reference presenti negli articoli) o presenti in troppi o troppo pochi documenti (parole come "and", "or", "if", ...).

Per ridurre la dimensione del vocabolario (per scremare i termini più importanti e ridurre i tempi di calcolo dovuto alla gestione di un numero così grande di informazioni) rimuoviamo allora i termini presenti meno di `term_min` volte in tutti gli articoli, e quelli che compaiono in troppi documenti, attraverso la funzione `pruned_vocabulary`.

La funzione `vocab_vectorize` del pacchetto `text2vec` trasforma il vocabolario ottenuto in un oggetto che, preso in input un testo t, restitusce il vettore v_t associato. Il procedimento illustrato per creare un *vectorizer* è implementato nel codice nella funzione `make_vectorizer`:

```
make_vectorizer(it_train, doc_prop_max, doc_prop_min, term_min, n)
```

che prende come parametri `doc_prop_max` e `doc_prop_min` (le proporzioni max e minime dei termini all'interno dei documenti) `term_min` (il numero minimo di volte che un termine deve comparire) e l'ampiezza `n` del più grande n-gramma considerato.

Applicando metodi diversi ho tarato diversamente i parametri della funzione `make_vectorizer`. Inoltre in tutti i modelli costruti ho sempre considerato anche di pre-processare i dati di input per rimuovere i simboli di latex e convertire tutto in minuscolo, usando la funzione `clean_string()`:

```
clean_string <- function (string_vec){
  string_vec = str_to_lower(string_vec)
  latex = c("\\mathbb", "\\bf", "\\mathcal", "\\mathbf", "\\dots")
  for (pt in latex){
    string_vec = gsub(pt, "", string_vec, fixed=TRUE)
  }
  return(string_vec)
}
```

Tuttavia nel modello finale questa funzione non è stata usata perché si è osservato che i risultati erano migliori senza pre-processare i testi.

A questo punto abbiamo ottenuto un vocabolario scremato delle parole presenti nei testi del training set, e possiamo vettorizzare i testi del test set e del validation set usando questo vocabolario e la funzione `create_dtm` del pacchetto `text2vec`.
Nel nostro caso ogni dato di input ha due testi, il titolo e l'abstract: trattiamo i due dati separatamente, creando un vocabolario a partire dai titoli dei testi di input e uno a partire dagli abstract dei testi di input.

```
it_train_ab = itoken(training$abstract, tokenizer = word_tokenizer)
tovec_ab = make_vectorizer(it_train_ab, doc_prop_max,
                                        doc_prop_min, term_min, n)
it_train_tit = itoken(training$title, tokenizer = word_tokenizer)
tovec_tit = make_vectorizer(it_train_tit, doc_prop_max,
                                        doc_prop_min, term_min, n)

dtm_train_ab = create_dtm(it_train_ab, tovec_ab$vectorizer)
dtm_train_tit = create_dtm(it_train_tit, tovec_tit$vectorizer)

it_val_ab = itoken(validation$abstract, tokenizer = word_tokenizer)
dtm_val_ab = create_dtm(it_val_ab, tovec_ab$vectorizer)
it_val_tit = itoken(validation$title, tokenizer = word_tokenizer)
dtm_val_tit = create_dtm(it_val_tit, tovec_tit$vectorizer)
```

2 Modelli provati

Le matrici create tramite il pacchetto `text2vec` sono matrici sparse di grandi dimensioni, per questo è necessario testare metodi che riescano a gestire un numero di fattori alto (in questo caso è pari a 6908) e matrici sparse (convertire i dati in matrici non sparse li rende molto difficili da usare per motivi comutazionali).

Inoltre il numero di label del training set è pari a 377 (quindi molto alto) e, come si vede dal plot della distribuzione dei label qui sotto, pochi label sono molto rappresentati, e invece altri sono presenti in pochi articoli (a volte anche uno solo).

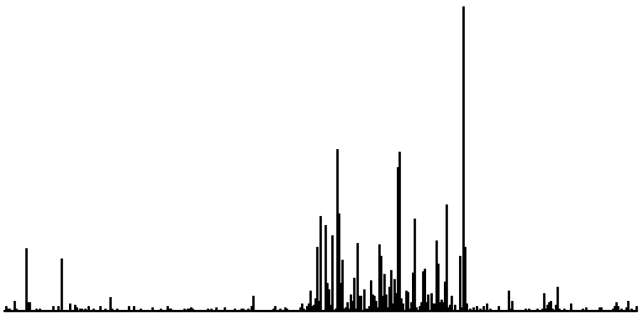

Figura 1: Numero di articoli del training set per MSC code

Ho quindi calibrato sul training set i seguenti metodi, che ritenevo adatti perché in grado di gestire classificazioni multiclasse e matrici sparse.

- k**-NN**: dato l'altissimo numero di fattori, e la "maledizione della dimensionalità" intriseca del metodo, non mi aspettavo grandi risultati, ma ho comunque testa il metodo direttamente sui dati e dopo averli proiettati sulle componenti principali.
- **SVM**: implementare una SVM sembra un metodo promettente poiché la classificazione data da questo algoritmo si basa sui margini di separazione dei dati e sembra non soffrire dell'alta dimensionalità del problema. Questo ci permette quindi di non dover ridurre molto i dati in input e poter mantenere più fattori.
- **Naive Bayes**: l'idea del classificatore Bayesiano multiclasse è quella di di predire la classe di un dato combinando le probabilità dei singoli fattori di appartenere ad ogni classe. Usarlo nell'analisi testuale significa decidere la classificazione del testo in base alle frequenze con cui i singoli termini che lo compongono compaiono nei vari argomenti, e anhc equesta sembra una strada promettente.

Dopo aver vettorizzato i dati di input di entrambi i set, ho selezionato solo il primo MSC code di ogni dato in input.

```
1 training$msc1 = sapply(training$msc, function(vec) vec[1])
2 training$msc1 = str_extract(training$msc1, "[:digit:][:digit:][:
    upper:][:digit:][:digit:]")
3 training$msc1 = factor(training$msc1)
```

Per il calcolo dell'errore ho utilizzato la funzione `AccuracyRate`, il cui codice è riportato di seguito: dato un vettore `pred` e una lista di vettori `test` lunga quanto il vettore restituisce la percentuale di valori predetti che appartengono al vettore corrispondente nella lista (la useremo per vedere quanti degli MSC predetti sono uno degli MSC assegnati all'articolo).

```
AccuracyRate <- function(pred, test){
  p = as.matrix(pred)
  er = 0
  for (i in 1:length(pred))
    { er = er + as.numeric(p[i] %in% test[[i]]) }
  return(100*(er/length(pred)))
}
```

Infine nel seguito per praticità indicheremo con `Xtrain` e `Xval` i dati di input del training e validation se: dati i vettori associati a titolo ed abstract associamo ad ogni articolo il vettore ottenuto unendo questi due.

```
Xtrain = cbind(dtm_train_tit,dtm_train_ab)
Xval = cbind(dtm_val_tit, dtm_val_ab)
```

2.1 kNN

Ho effettuato varie prove con il metodo k-NN, variando la costruzione del vocabolario, ed i parametri migliori per costruire il dizionario sono risultati `term_min=3`, `n=1` (ovvero considerando solo parole singole), `doc_prop_max` e `doc_prop_min` definiti come di seguito.

```
v = rep(0,length(levels(training$msc1)) )
for (i in 1:length(v)){v[i]=sum(training$msc1==training$msc1[i])/N}
doc_prop_max = max(v)
doc_prop_min = min(v)
```

Osserviamo che:

- i label meno rappresentati sono 195 e sono presenti in solo 1 articolo;
- il label più rappresentato è "60K35" ed è l'MSC code di 237 articoli;

Quindi in questo caso `doc_prop_min` non taglia nessun termine, mentre `doc_prop_max` rimuove tutti i termini presenti in più di 261 articoli (per il principio dei cassetti un tale termine apparterrà almeno a due articoli con classificazione diversa).

Ho applicato per prima cosa k-NN direttamente ai dati iniziali, e testato diversi valori di k:

```
knn.tune<- list(k = c(), preds = as.character(c()),
                  acc = c(), accAll = c())
kval = c(1,2, 5, 20, 50)
for (k in kval){
  knn.tune$k = c(knn.tune$k, k)
  knn_pred = knn(train = Xtrain, test = Xval, cl = Ytrain, k = k)
  knn.tune$preds = cbind(knn.tune$preds, as.character(knn_pred))
  accAll <-AccuracyRate(knn_pred, validation$msc)
  knn.tune$accAll = c(knn.tune$accAll, accAll)
}
```

Il risultato migliore una precisione del 20.6%, ottenuta per $k = 20$, ma osserviamo che computazionalmente le operazioni sono impegnative (circa 100 secondi a iterazione).

Per cercare di far fronte al problema della dimensionalità ed estrapolare dai dati le informazioni principali, ho ridotto i dati attraverso PCA ed eseguito di nuovo k-NN sui dati ridotti.
Per usare PCA sulla matrice `Xtrain` (sparsa e con un numero di fattori molto più grande del numero di dati), ho utilizzato la funzione `nsprcomp` dell'omonima libreria.

```
library(nsprcomp)
pca = nsprcomp(Xtrain, ncomp = 19, scale = TRUE, nrestart = 3)
Xtrain.pca = pca$x
Xval.pca = predict(pca, Xval)

for (k in c(10,20,50,100)){
  acc = c()
  for (i in c(1:10)){
    knn.pca = knn(train = Xtrain.pca, test = Xval.pca, cl = Ytrain,
        k = k)
    acc = c(acc,AccuracyRate(knn.pca, validation$msc))
  }
  print(mean(acc))
}
```

Ho testato la funzione con `ncomp` (numero di componenti principali) fra 5 e 30 (e già in questi casi il tempo richiesto per il calcolo era di circa 500 secondi), ottenendo con 19 componenti e $k = 50$ una precisione circa del 30%, migliore rispetto al caso non ridotto ma ancora poco soddisfacente.

2.2 SVM

Anche per le Support Vector Machines ho osservato che i risultati ottimali si ottenevano con un dizionario costruito come spiegato nel k-NN, ed in questo caso abbiamo ottenuto risultati dedcisamente migliori.

Ho provato diverse combinazioni di parametri e kernel, per cui ho stimato l'errore di validazione usando la funzione `tune_svm`, di cui riporto una parte del codice qui sotto (in fondo allo script su Colab ho riportato il codice integrale).

```
tunesvm = function(Xtrain,Ytrain,Xval,YvalAll,knType,par){
  svmtuner = list()
  k = 3
  if(isTRUE(knType == "linear")){...}
  if(isTRUE(knType == "polynomial")){
    if(is.null(par$degree)){deg = 2;}else{deg =par$degree}
    if(is.null(par$gamma)){gamma = 0.1}else{gamma=par$gamma}
    if(is.null(par$cost)){cost = 1}else{cost=par$cost}
    if(is.null(par$coef0)){ceof0 = 1}else{coef0=par$coef0}

```

```
    for (d in deg){
      svmtuner$degree = c(svmtuner$degree, rep(d, length(gamma)*
          length(cost)*length(coef0) ) )
      for (g in gamma){
        svmtuner$gamma = c(svmtuner$gamma, rep(g, length(cost)*
            length(coef0) ) )
        for (c in cost){
          svmtuner$cost = c(svmtuner$cost, rep(c,length(coef0)) )
          for (c0 in coef0){
            svmtuner$coef0 = c(svmtuner$coef0, c0)
            preds <- c() accAll <- c() acc <- c()
            for (j in 1:k){
              svmfit = svm(y = Ytrain,x = Xtrain, kernel = knType,
                  cost = c, degree = d, gamma = g, coef0 = c0)
              preds = cbind(preds, as.character(predict(svmfit, Xval
                  )) )
              accAll = c(accAll, AccuracyRate(preds[,j],YvalAll))
            }
            prediction = apply(preds, 1, MostCommon)
            svmtuner$preds = cbind(svmtuner$preds, prediction)
            svmtuner$accAll = c(svmtuner$accAll, mean(accAll))
          } } } }
    l = which.max(svmtuner$accAll)
    svmtuner$bestmodel = ...
    ##salva in bestmodel le informazioni (parametri, accuratezza e
        predittore)
    ##del modello migliore
    return(svmtuner)
  }
  if(isTRUE(knType == "radial")){...}
}
```

Dati dei parametri per la SVM in input, la funzione `tune_svm` calibra un metodo svm sul training set per ogni combinazione dei parametri in input e testa l'accuratezza sul validation set con una 3-fold. Per determinare l'errore e i predittori di una combianzione di parametri utilizza la funzione `MostCommon`, così definita:

```
MostCommon <- function(x){
  #dato un vettore x trova il valore più rappresentato
  names(which.max(table(x)))
}
```

Nella tabella seguente sono riportati i test effettuati variando il kernel ed i parametri.

Kernel	Parameters	Accuracy Rate
Polinomyal	`gamma = 0.1, cost = 2, coef0 = 1, deg = 1`	52.3%
Linear	`cost = 3`	52.2%
Radial	`gamma = 0.01, cost = 17`	50%

Effettuando ulteriori test con kernel polinomiale di grado 1 (che si è rivelato il migliore), otteniamo una precisione del 52.6% usando come parametri `gamma = 0.01, cost = 7, coef0 = 1`.
Osserviamo però che ogni chiamata della funzione `svm` sull training set è ancora computazionale pesante, e impega circa 50 secondi.

3 Il metodo migliore: Naive Bayes

Siano $C = \{c_1, \dots, c_K\}$ i label, $\{a_1, \dots, a_N\}$ gli articoli del training set e $V = \{w_1, \dots, w_T\}$ il vocabolario: come visto ogni articolo a sarà un vettore $v_a = (v_i^a)$.
Come visto a lezione, il Naive Bayes Classifier assegna ad un articolo a il label

$$\operatorname*{argmax}_{c_k \in C} \mathbb{P}(C_k|a) = \operatorname*{argmax}_{c_k \in C} \mathbb{P}(a|c_k)\mathbb{P}(c_k)$$

dove supponiamo che, dato il training set, valga:

- $\mathbb{P}(c_k) = \#\{a_i \text{ con label } c_k\}/N$
- le probabilità degli input, condizionati gli output, siano indipendenti. Ovvero, detta

  $$\theta_{t,k} = \#\{a_i \text{ con label } c_k \text{ che contengono } w_t\}/\#\{a_i \text{ con label } c_k\}$$

  la probabilità condizionata che un articolo con label c_k contenga la parola w_t, supponiamo che

  $$\mathbb{P}(a|c_k) = \mathbb{P}(v_a|c_k) = \prod_{t=1}^{T} \mathbb{P}(v_t^{(a)}|c_k) = \prod_{t=1}^{T} \theta_{t,k}^{v_t^{(a)}}.$$

Osserviamo però che, se una parola w_t non compare mai in un testo di tipo c_k nel training set, $\theta_{t,k} = 0$: quindi se un articolo a contiene la parola w_t tutte le informazioni sulla probabilità che le altre parole siano di tipo c_k sono "cancellate" da $\theta_{t,k}$ e $\mathbb{P}(a|c_k) = 0$.

Per far fronte a questo problema, consideriamo dei $\widehat{\theta_{t,k}}$ modificati, dati da

$$\widehat{\theta_{t,k}} = \frac{\#\{a_i \text{ con label } c_k \text{ che contengono } w_t\} + \alpha}{\#\{a_i \text{ con label } c_k\} + \alpha T}$$

dove α è un parametro positivo fissato (solitamente piccolo rispetto a T, cardinalità del vocabolario).
In questo modo $0 < \widehat{\theta_{t,k}} \leq 1$, la somma $\sum_{t=1}^{T} \widehat{\theta_{t,k}} = 1$, e, se w_t non compare mai in un testo di tipo c_k, $\widehat{\theta_{t,k}} = 1/(T + \frac{1}{\alpha}\{a_i \text{ con label } c_k\})$ è vicino a 0 ma non nullo.

Per il classificatore Naive Bayes ho usato la funzione `multinomial_naive_bayes` della libreria `naivebayes`: questa funzione prende come parametro prende anche `laplace`, il parametro α introdotto sopra.

La funzione usata, ottimizzata per l'uso delle matrici sparse, è molto veloce e permette di effettuare numerosi test e variare bene i parametri. In particolare ho potuto testare molte combinazioni dei paramentri per la creazione del vocabolario, osservando che i risultati migliori si ottenevano chiamando:

```
tovec_ab = make_vectorizer(it_train_ab, 0.19, 0, 10, 2)
tovec_tit = make_vectorizer(it_train_tit, 0.19, 0, 10, 2)
```

Una volta creati i vettori `Xtrain` e `Xtval` a partire dai vectorizer creati sopra, testiamo il Naive bayes per diversi valori di α:

```
acc <- c()
lap = c(0.2*(1:8), 1.6+0.05*(1:8), 2+0.2*(1:5))
for (l in lap){
  t1 = Sys.time()
  mnb = multinomial_naive_bayes(Xtrain, Ytrain, laplace = l)
  preds = predict(mnb, newdata = Xval, type = "class")
  acc <- c(acc,AccuracyRate(preds, YvalAll))
}
```

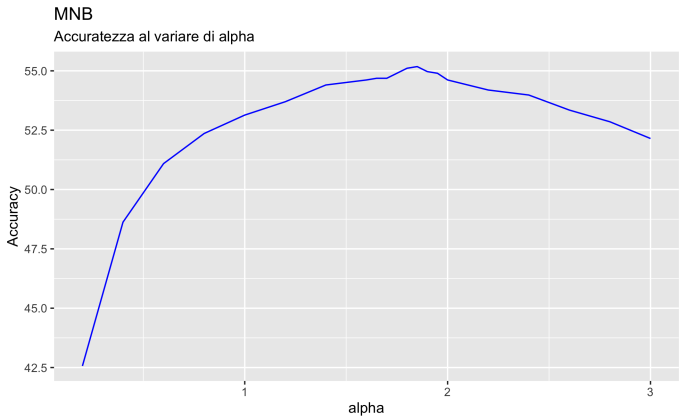

Figura 2: precisione del MNB al variare della correzione

Come si osserva quindi la precisione migliore si ottiene per $\alpha = 1.85$ ed è pari al 55.2%, risultato migliore (anche se solo leggermente) di quello ottenuto con SVM. Anche se la differenza con il risultato ottenuto da SVM è poca, credo che Naive Bayes possa essere considerato migliore soprattutto per i tempi computazionali decisamente ridotti (1 secondo per tarare il metodo e predire i risulti), quindi utilizzerò questo come metodo per predire il blind test set.

4 Possibili miglioramenti

Purtroppo, nonostante i numerosi tentativi fatti, l'accuratezza ottenuta è pari al 55%, risultato poco soddisfacente. Il seguente grafico mostra, al variare dell'MSC-code C, il numero di articoli stimati con C per cui la stima era esatta (in verde) o scrorretta (in rosso). Inoltre la linea nera rappresenta l'altezza del verde per avereuna precisione del 60% in ogni barra e i punti viola rappresentano il numero di articoli del training set con il rispettivo MSC code (non ho rappresentato tutti i codici, ma solo quelli presenti almeno 5 volte nel validation set).

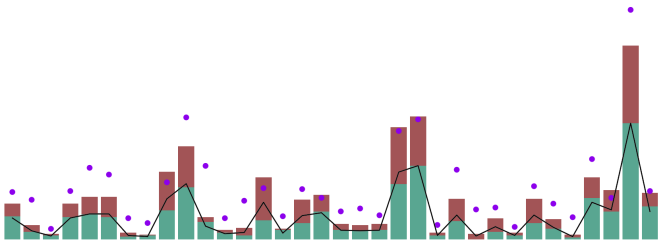

Figura 3: Numero di articoli correttamente e scorrettamente stimati dato l'MSC Code

Come potevamo aspettarci, gli MSC per cui la precisione è stata magiore del 60% sono quelli per i quali avevamo più articoli del training set, quindi possiamo ipotizzare che uno dei motivi per cui il risultato ottenuto è poco soddisfacente è che molti dei label del training set sono poco rappresentati. Riporto di seguito uno dei tentativi effettuati per cercare di migliorare il risultato ottenuto che mi era sembrato promettente e che probabilmente, con un training set più ricco e un computer con una potenza di calcolo maggiore, potrebbe dare risultati migliori.
L'osservazione chiave è che le tre parti da cui è composto l'MSC code (numero primario, lettera secondaria e numero terziario) sono dipendenti fra di loro: la lettera è il subargomento relativo all'argomento dato dal numero primario. Una possibilità per prederire l'MSC code potrebbe essere quindi quella di predirlo in più fasi, ovvero:

- A partire dal training set completo predire il *numero primario*;
- per ogni numero primario n predetto considerare il sotto-training set composto dai soli articoli con numero primario n e implementare un metodo di predizione per la *lettera secondaria* a partire solo dal sotto-traing set scelto;
- come al punto precedente, implementare un metodo di predizione per il *numero terziario* a partire solo dal sotto-traing set costituito dagli articoli con numero primario e lettera fissati.

Il vantaggio di un approccio come questo potrebbe essere dato dal fatto che "specializzare" il training set (eventualmente creando nuovamente i vocabolari, in modo da far emergere le parole più significative di un determinato ambito) potrebbe portare a dare più peso nella predizione a delle caratteristiche specifiche, che nella predizione globale potrebbero perdersi.

Provando ad implementare questo metodo ho visto che in effetti l'accuratezza al primo passo arriva fino al 90% di precisione, sia con SVM che con Naive Bayes (andando oltre il predittore diventa costantemente "60", che in effetti è fra gli MSC code del 95% dei testi). Purtroppo però, probabilmente perché il numero di testi di molti numeri principali è troppo piccolo, la precisione finale viene intorno al 40%.