Algoritmo “*UncOptCost*”

Un sistema di crowdsourcing ottimale

Emanuele Tusoni, Valerio De Matteis, Danilo Ponti

L’algoritmo si prefigge lo scopo di minimizzare il numero di fasi e il numero di domande da porre agli utenti in un sistema di crowdsourcing.

Il fine ultimo è quello di trovare un numero arbitrario di oggetti “**O”** che soddisfino una certa condizione “**C”** nel minor tempo (fasi) e costo (domande) possibile ponendo delle domande a dei worker appositamente assunti.

L’assunzione fondamentale su cui si basa il sistema è che alcuni problemi sono più facilmente comprensibili e risolvibili dall’uomo rispetto ad una macchina.

In questo caso la strategia consiste nel porre ad ogni fase un numero preciso di domande in parallelo ai vari worker su degli oggetti precedentemente scelti in base ad alcuni criteri specifici.

La scelta degli oggetti è guidata da una matrice dei costi basata sul calcolo delle probabilità condizionate, dove ogni punto ha come ordinata il numero di risposte “si” e come ascissa quello delle risposte “no”.

Il numero di questi oggetti per ogni fase è equivalente alla differenza tra gli oggetti richiesti in partenza e quelli che sono stati già verificati come idonei, ovvero gli oggetti che ancora ci mancano per terminare il processo.

Le domande da porre in parallelo ai worker, infine, sono poste in una coda e il loro numero è uguale alla sommatoria per ogni oggetto del minimo numero di domande che ci permette di classificarli definitivamente, sia in positivo che in negativo.

La seconda importante assunzione è che le risposte date dai worker non sono sempre esatte: in questo caso viene stimato un error rate umano (20% ad esempio) che va ad influenzare anche il calcolo della matrice dei costi.

A tal proposito è necessario un sistema efficiente che ci permetta di discriminare worker maliziosi da quelli semplicemente disattenti, in modo da espellerli dal processo prima che degradino il risultato finale.

1. Implementazione dell'algoritmo

L'implementazione dell'algoritmo principale della gestione delle domande e degli oggetti è la seguente:

```
173  def unc_opt_cost_algorithm(Items, K1):
174
175      precision = 0
176      global ALFA
177      global cont_phase
178      ALFA = m/2
179      founds = list()
180      itemsToTest = list()
181      itemsToTest = Items[:]
182      total_answers = {}
183      for x in range(len(Items)):
184          total_answers[Items[x]] = (0, 0)
185      pointCostMap = {}
186      ComputeYForEachPoint(pointCostMap)
187
188      if(m>1):
189          while(not RightValueALFA(pointCostMap)):
190              ComputeYForEachPoint(pointCostMap)
191
192      while len(founds) < K1:
193
194          if(len(itemsToTest)<(K1-len(founds))):
195              break
196          cont_phase += 1
197          ItemsLowestCost = getLowestCostItems(K1 - len(founds), total_answers, pointCostMap, itemsToTest)
198          CQ = list()
199          for item in ItemsLowestCost:
200              fewestYes = fewestYesQuestionsToZero(item, total_answers)
201              fewestNo = fewestNoQuestionsToAlfa(item, total_answers)
202              q = question
203              for x in range(min(fewestYes, fewestNo)):
204                  CQ.append((item, question))
205          partial_answers = crowdSourcingSystem(CQ)
206          updateAnswers(partial_answers, total_answers)
207          for item in ItemsLowestCost:
208              rectangularStrategy(total_answers[item], item, itemsToTest, founds)
```

La variabile *ALFA* rappresenta il valore inziale che viene dato al punto di coordinate *(0,0)*, ovvero agli oggetti che non hanno ancora ricevuto risposte né positive né negative. Questo valore in realtà viene successivamente aggiustato in relazione alla mappa dei costi generale *"pointCostMap"*, che, a sua volta si basa sul calcolo di un costo per ogni punto di una matrice *m*m*, con m opportunamente scelto. La funzione incaricata del calcolo è la seguente:

```
337  def Y(n1,n2,pointCostMap):
338      if(n1 == 0 and n2 ==0):
339          return ALFA
340
341      elif(n1 != m-1 and n2!= m-1):
342
343          return min(ALFA, ((p1(n1, n2) * Y(n1+1,n2,pointCostMap)) + (p0(n1, n2) * Y(n1,n2+1,pointCostMap)) + 1))
344
345      elif(n1 == m-1):
346          return 0
347      else:
348          return ALFA
```

Y(n1, n2) calcola un costo che dipende solo dal punto *(n1,n2)*, dalla selettività del predicato e dall'error rate dei worker che rispondono. La funziona associa al punto *(0,0)* un costo *ALFA*, nei punti in cui abbiamo ricevuto tutte risposte positive è pari a 0, mentre al contrario nei punti in cui abbiamo tutte risposte negative questo è di nuovo uguale ad *ALFA*.

Il calcolo di un punto generale della strategia è probabilistico, sfruttando altre due funzioni: *"p1(n1, n2)"* e *"p0(n1, n2)"*, ovvero rispettivamente la probabilità di ricevere una risposta *"Yes"* nel punto *(n1, n2)*, e la probabilità di ricevere una risposta "No" trovandosi nel punto *(n1, n2)*.

```
326 def p1(n1,n2):
327     return (Pr3(1,n1, n2) * Pr2('YES',1)) + (Pr3(0,n1, n2) * Pr2('YES',0))
328
329
330
331 def p0(n1, n2):
332     return (Pr3(1,n1, n2) * Pr2('NO',1)) + (Pr3(0, n1, n2) * Pr2('NO',0))
```

Nelle funzioni precedentemente illustrate *Pr3(1| (n1, n2))* è la probabilità che l'oggetto in esame soddisfi il predicato a fronte del fatto che ci si trovi nel punto *(n1,n2)*, similmente *Pr3(0| (n1,n2))* è la probabilità che invece l'oggetto non lo soddisfi. L'implementazione di questa funzione è la seguente:

```
310 def Pr3(bool,n1,n2):
311     if(bool == 1):
312         return pow((1 - ERROR_RATE), n1) * pow(ERROR_RATE, n2)
313     else:
314         return pow(ERROR_RATE, n1) * pow((1 - ERROR_RATE), n2)
315
```

Le funzione *Pr2*, infine, esprime proprio la probabilità che la prossima risposta sia "Yes" oppure "No" condizionata dal fatto che l'oggetto soddisfi o meno il predicato. Questo può essere interpretato come l'error rate di un worker, che può rispondere No (o Yes) anche se l'oggetto in realtà soddisfa il predicato (o se non lo soddisfa). Questo deve essere preso in considerazione proprio perché si è assunto che i worker possano dare risposte errate, che è sicuramente lo scenario più realistico.

```
318 def Pr2(answer,bool):
319     if( (answer == 'YES' and bool == 1) or (answer == 'NO' and bool == 0)):
320         return (1-ERROR_RATE)
321     else:
322         return ERROR_RATE
```

Una volta calcolata la matrice dei costi, il resto dell'algoritmo si occupa di selezionare, per ogni fase, tanti oggetti quanti ancora ne servono tra quelli che presentano i costi minori proprio a fronte delle risposte che hanno ricevuto precedentemente, grazie alla funzione *"getLowestCostItems"*.

Una volta fatto questo, per ogni oggetto viene generato un numero di domande pari al minimo tra le domande che servono per determinare che l'oggetto soddisfi il predicato e quelle che servono per determinare che l'oggetto non lo soddisfi e le si mette tutte quante in una coda.

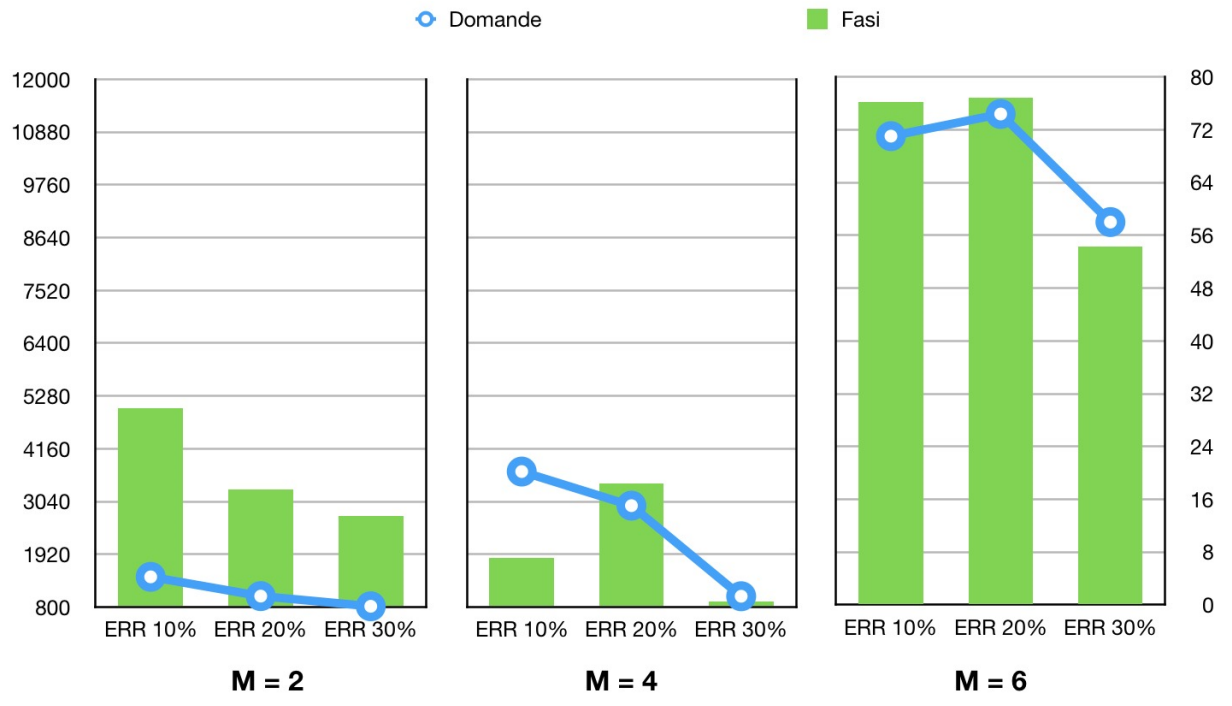
Le domande così generate sono poi successivamente poste ai worker in modo parallelo, e le risposte ricevute vanno ad aggiungersi a quelle già collezionate.

A seconda della strategia scelta, infine, viene determinato se gli oggetti in esame possono essere considerati idonei oppure no, e l'algoritmo prosegue finché non è stato trovato il numero di oggetti idonei k1 specificati.

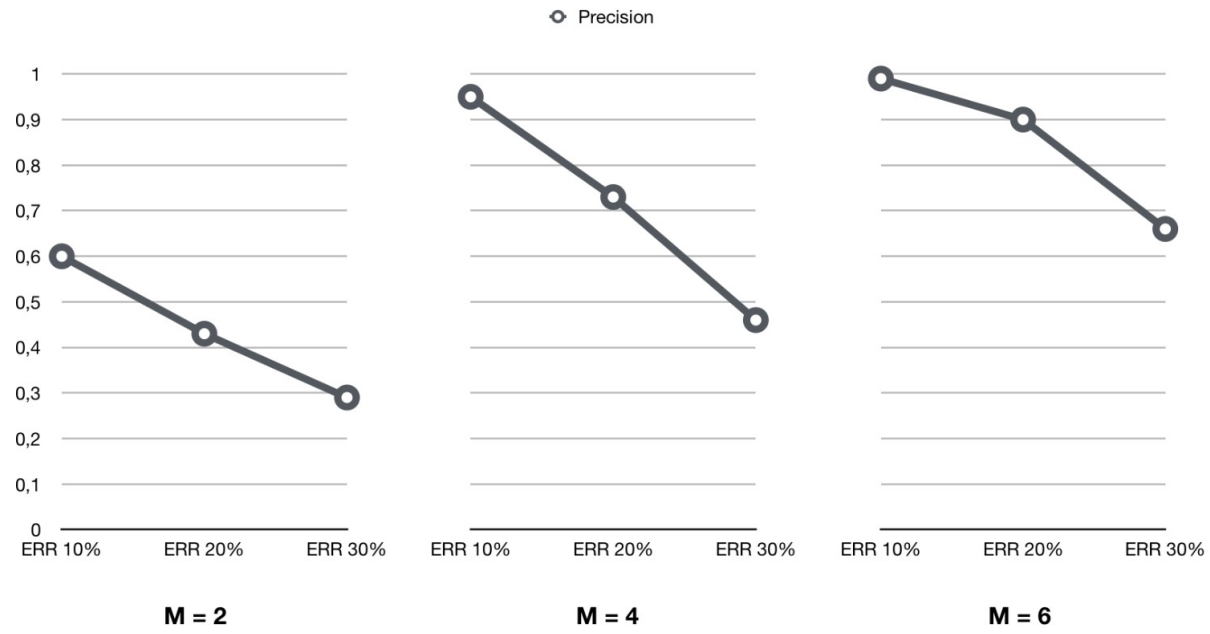
2. Test UncOptCost

I nostri test si sono concentrati nel confrontare il numero di domande, il numero di fasi e la precisione di ogni iterazione dell'algoritmo al variare dell'error rate generale dei worker, al valore di *m* scelto ed infine alla percentuale di oggetti che soddisfano il requisito richiesto su un dataset di 10.000 oggetti.

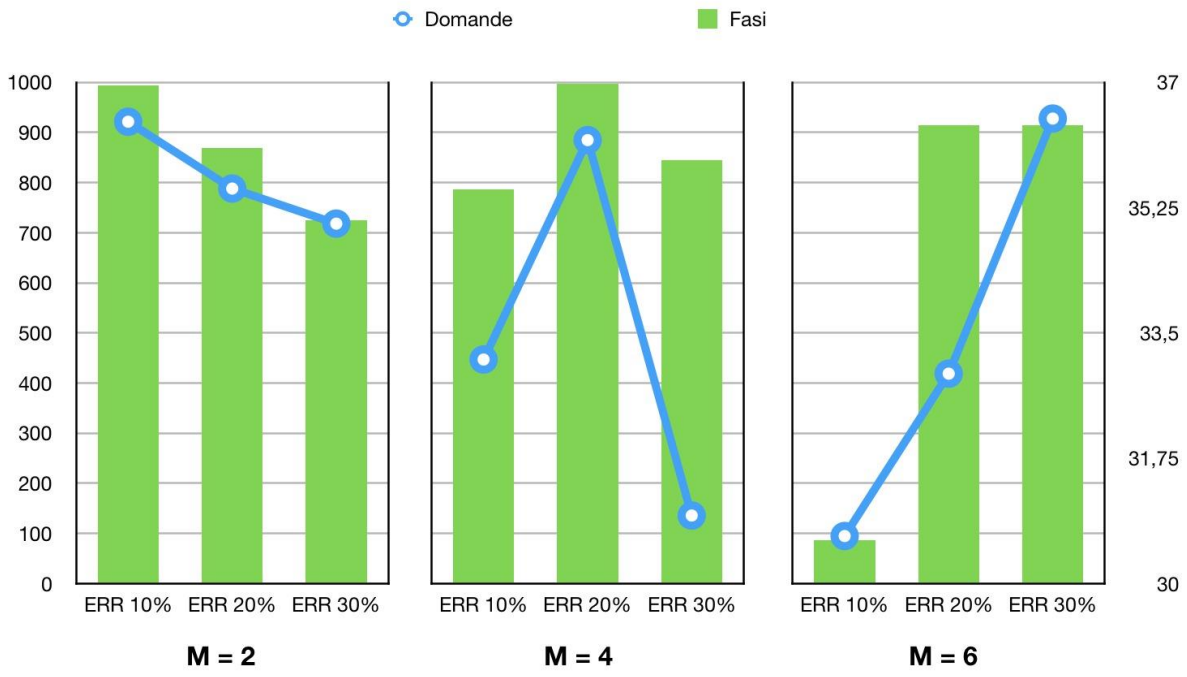
Nel primo caso abbiamo inserito un 15% di oggetti che soddisfano il predicato rispetto all'intero dataset e abbiamo variato l'error rate generale degli utenti e *m* andando a misurare la media delle domande e delle fasi su 10 iterazioni.

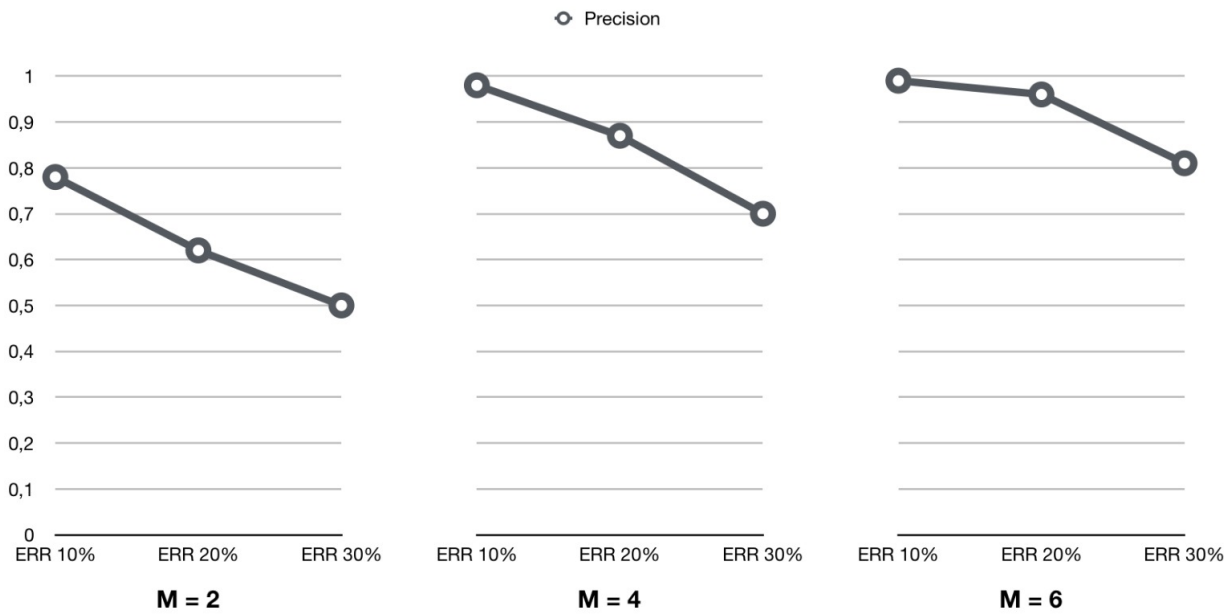


Con gli stessi parametri abbiamo poi calcolato la precisione del risultato finale come il rapporto tra il numero di oggetti che davvero hanno soddisfatto il predicato e il numero di oggetti totali.



Questi invece sono i risultati con un 30% di oggetti che soddisfano il predicato richiesto all'interno del dataset:





3. Implementazione del sistema anti spammers

Per implementare il sistema anti spammers ci siamo basati su un'assunzione in particolare: durante tutto il processo la maggior parte dei worker tenderà a rispondere in modo esatto mentre gli spammer rappresenteranno una minoranza. Questo ci permette di arrivare ad alcune conclusioni: mediamente una grande percentuale di utenti tenderà a rispondere allo stesso modo mentre un'altra piccola percentuale avrà un comportamento che potremmo definire anomalo.

L'idea principale è che quando questo comportamento è particolarmente evidente possiamo andare ad operare una politica di incremento o decremento di un punteggio associato all'utente a seconda del comportamento rilevato. Quindi se un'alta percentuale di utenti (ad esempio l'80-90%) ha risposto al medesimo modo mentre una piccola percentuale ha risposto diversamente, andremo ad alzare il punteggio degli utenti in minoranza, mentre, al contrario, andremo a diminuire quello degli utenti in maggioranza. Il punteggio in questo caso può essere visto come una penalità che ci permette di "tenere d'occhio" gli utenti che si discostano largamente dagli altri. Quando il punteggio di un utente raggiunge una certa soglia, questo viene estromesso dal processo e non gli saranno più rivolte domande.

In generale la soglia, l'incremento e il decremento della penalità possono essere regolati a seconda del task o della politica: una diminuzione della soglia così come l'incremento della penalità equivarrà ad una politica di tolleranza agli errori degli utenti più stringente.

È evidente che il sistema si basa anche su una seconda assunzione, ovvero che ci sia un numero minimo di utenti e di risposte richieste per un singolo oggetto. Questo è ragionevole se pensiamo che si possa optare per questa strategia nel momento in cui vogliamo che vengano minimizzati i costi necessari (magari già abbastanza alti). Questo algoritmo, infatti, non richiede di porre altre domande se non quelle strettamente necessarie richieste dall' "UncOptCost", laddove si potrebbe pensare di sottoporre al worker delle domande di sicurezza di cui sappiamo già la risposta per capire se l'utente sbaglia intenzionalmente.

4. Test dell’algoritmo anti spammers

In questo caso i nostri sforzi per testare il sistema si sono concentrati sul calcolo della percentuale di rilevazione degli spammer variando l’error rate generale di tutti gli tenti mantenendo quello degli spammer al 50% (ovvero una scelta completamente casuale tra due opzioni).

Nello specifico i valori calcolati fanno riferimento ad una media su 50 iterazioni dell’algoritmo con 7 utenti, prima con la presenza di 1 spammer, poi con 2 ed infine con 3, variando l’error rate generale di tutti gli altri utenti.