Implementazione di un RS basato su Collaborative Filtering

Progetto di Intelligent Web

Università degli Studi di Napoli Federico II
Corso di Laurea Magistrale in Informatica
Docente: prof. Luigi Sauro

Ivano Matrisciano
N97/392

Alessandro Quirile
N97/402

febbraio 2023

1. Introduzione

Questo documento descrive il funzionamento e gli aspetti implementativi dei principali algoritmi di un sistema di raccomandazione (RS) in Julia[JU] basato su Collaborative Filtering utilizzando il dataset MovieLens-Small[DS]. Viene applicata la k-fold cross validation come tecnica di validazione per la model selection.

Realizzare una recommender system che, mediante una tecnica di collaborative filtering, prende in input una URM (model building set) produce delle stime di ranking da confrontare con un test set.

1.2 Articolo scientifico di riferimento

L'articolo scientifico di riferimento[PAP] introduce una nuova metrica che migliora le prestazioni di un RS in termini di accuratezza e di predizioni perfette, intese come predizioni uguali al target (a meno di una tolleranza per gestire confronti in virgola mobile) quando lo span di voti esprimibili è sufficientemente piccolo – ad esempio da 1 a 5. Approcci memory-based usano tipicamente l'indice di Pearson[PEA] per determinare la similarità tra due utenti x e y del RS, intesi come vettori dei voti che ciascuno può esprimere per ogni item del dataset. Sfortunatamente, da un punto di vista puramente teorico, l'indice di Pearson può essere applicato – con garanzie di successo – soltanto quando le seguenti condizioni sono verificate:

- x e y sono variabili aleatorie linearmente dipendenti
- x e y sono variabili aleatorie continue
- x e y sono variabili aleatorie campionate da una gaussiana

Queste condizioni non sono normalmente verificate in un RS. Di contro, l'indice di Pearson è ancora la metrica più utilizzata in questo contesto perché, empiricamente, produce discreti risultati[MET][NCF]. L'obiettivo è trovare una metrica alternativa che ha un fondamento teorico sufficientemente robusto da poter essere applicato ad un RS migliorando le prestazioni dell'indice di Pearson.

2. Analisi teorica

2.1 Analisi statistica del dataset

Innanzitutto descriviamo statisticamente i voti del dataset attraverso indici di posizione (media, moda e mediana), indici di dispersione (deviazione standard) ed indici di forma (skewness). I risultati ottenuti in questa fase giustificano alcune scelte progettuali future.

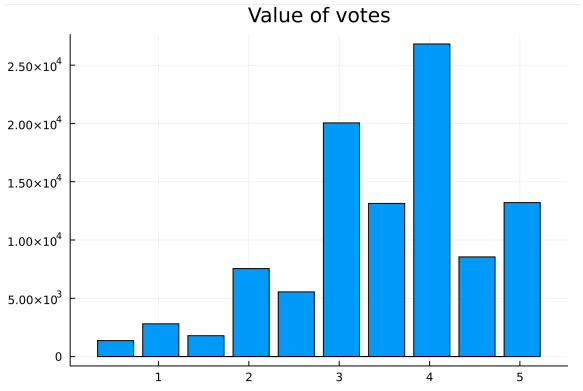


Fig 1. Distribuzione dei voti

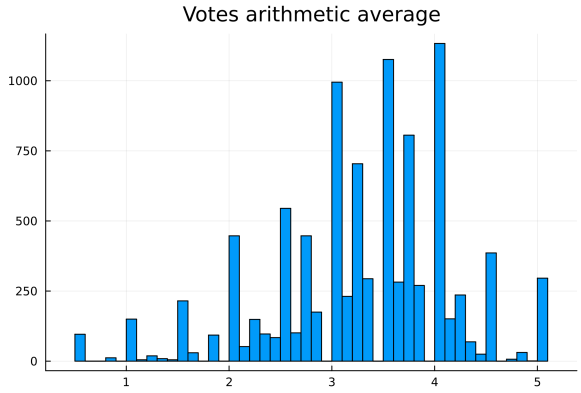


Fig 2. Media aritmetica dei voti

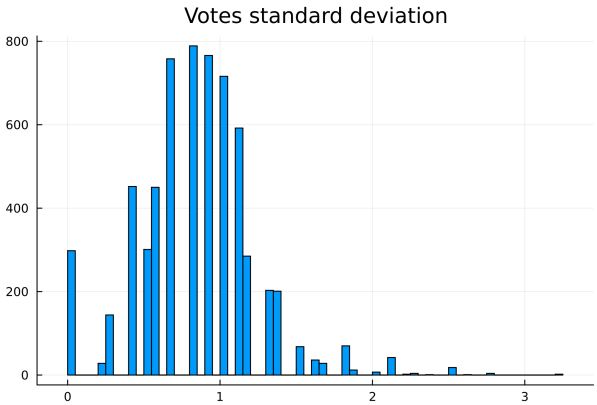


Fig 3. Deviazione standard dei voti

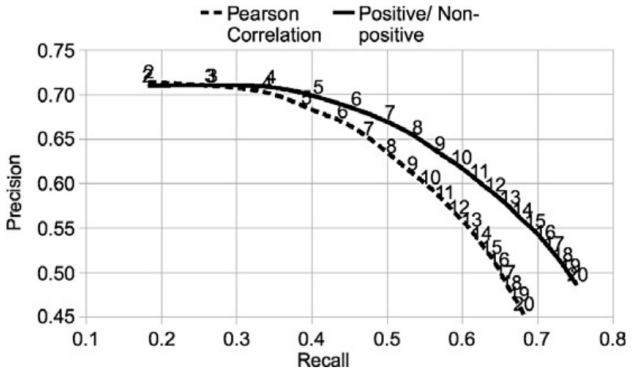
Si osserva che:

1. Lo span di voti esprimibili è l'intervallo $[0.5; 5]$ per un totale di 10 voti esprimibili, differentemente dalle vecchie versioni del dataset in cui i voti esprimibili erano soltanto 1, 2, 3, 4 e 5.
2. Gli utenti tendono a recensire un film solo quando è stato gradito (c'è maggiore densità nella parte a destra della distribuzione)

3. Pochissimi utenti sono propensi a dare un voto quando un film non è stato gradito
4. Il voto espresso con più frequenza è 4 (moda campionaria)
5. La mediana campionaria è 3½
6. I voti si distribuiscono in modo asimmetrico, con una coda a sinistra (skewness pari a -0.6)
7. I voti si discostano di molto poco (circa ½ punto) rispetto alla media
8. Solo una piccola parte di film viene valutata frequentemente a discapito dei film meno popolari che invece sono valutati raramente (long-tail problem)

2.2 Informazioni non-numeriche: l'indice di Jaccard

Nel processo di raccomandazione (non in quello di predizione), le informazioni puramente numeriche sembrano perdere rilevanza: c'è poca differenza qualitativa, ad esempio, tra un film recensito 4 ad uno recensito 4½: occorre trovare un modo per determinare la similarità tra due utenti prescindendo dai specifici valori dei loro voti: associamo ai voti 4, 4½ e 5 l'etichetta *Positivi* (P) e agli altri l'etichetta *Negativi* (N). Fatto ciò si individua un valore soglia θ=4 che separa gli elementi etichettati P da quelli etichettati N. I valori etichettati P verranno tutti considerati come "rilevanti" per l'utente in fase di raccomandazione. Come si nota dalla Figura 3 del paper[PAP] di fianco riportata, la discretizzazione non solo non peggiora precision e recall ma, addirittura, le migliora specialmente col crescere del numero di raccomandazioni. Queste considerazioni suggeriscono che il processo di raccomandazione trascende dall'esatta informazione numerica, quando lo span di voti esprimibili è sufficientemente piccolo. Bisogna trovare, però, un modo per determinare la similarità tra due utenti tenendo in considerazione anche il numero di voti espressi: un'alta similarità tra due utenti che hanno espresso voti per un numero esiguo di film non può essere credibile: la discretizzazione di cui sopra va estesa attraverso l'indice di Jaccard che tiene in considerazione la proporzione tra il numero di film votati in comune da due utenti rispetto al totale. Se denotiamo con r_x i rating dell'utente x, allora



$$\forall x, y \in Users \quad Jaccard(x,y) = (r_x \cap r_y) / (r_x \cup r_y)$$

2.3 Informazioni numeriche: la MSD

La misura di Jaccard può essere efficacemente complementata con una misura di distanza basata sulla geometria euclidea: la Mean Squared Differences (MSD). Le metriche tipicamente utilizzate (ad esempio similarità coseno o Spearman) mostrano, chi più e chi meno, le stesse

problematiche della metrica di Pearson. Empiricamente si è osservato che la MSD offre risultati interessanti rispetto alle altre metriche, tranne che nella coverage[MET] che viene definita come la capacità di un RS di generare raccomandazioni per un numero significativo di utenti o item: sistemi con un'elevata copertura saranno in grado di fornire raccomandazioni personalizzate e di qualità agli utenti. Inoltre la MSD tende a non considerare la quantità di film votati in comune. Formalmente $\forall x, y \in Users$ e $\forall 1 \leq i \leq I = |Items|$ definiamo:

$$r_x^i \in \{[0.5;\ 5]\} \cup \{missing\}$$

$$d_{x,y} = (d^1_{x,y},\ d^2_{x,y},\ \ldots,\ d^I_{x,y}) \text{ dove}$$

$$d^i_{x,y} = (r_x^i - r_y^i)^2 \quad se \quad r_x^i,\ r_x^i \neq missing;\ altrimenti\ missing$$

$$MSD(x, y) = \overline{d_{x,y}}$$

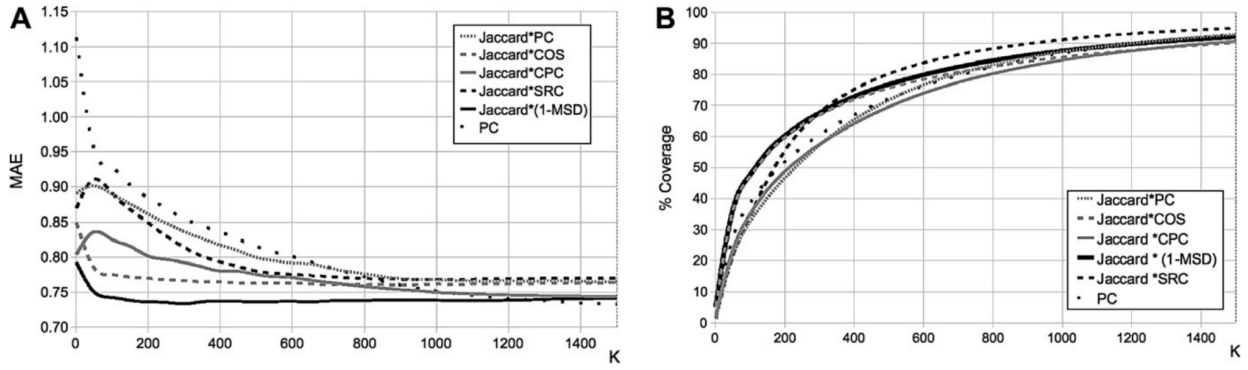
dove $\overline{d_{x,y}}$ rappresenta la media dei voti significativi (non missing) del vettore $d_{x,y}$

2.4 Formalizzazione della nuova metrica

Riassumendo, quando lo span di voti esprimibili è sufficientemente piccolo:

- Jaccard ci dà informazioni non-numeriche e diminuisce notevolmente la MAE
- MSD ci dà informazioni numeriche e dà ottimi risultati tranne che per la coverage

Combinando tali misure tra loro otteniamo una nuova metrica di similarità che mitiga i limiti di entrambe, come illustrato nella Figura 5 del paper[PAP] di seguito riportata.



Come si nota, la combinazione Jaccard&MSD abbatte notevolmente la MAE rispetto alle altre combinazioni Jaccard&Pearson, Jaccard&Cosine, Jaccard&ConstrainedPearson, Jaccard&Spearman e anche rispetto alla misura di Pearson usata da sola, specialmente per una taglia di vicinato $K \leq 1200$. Similmente anche la coverage migliora, seppur di poco, specialmente per $K \leq 300$, dopo di che risulterebbe ottimale la combinazione Jaccard&Spearman.

Un'altra considerazione da fare è tenere a mente è (l'enorme) sparsità della URM: i voti mancanti non andranno considerati nel calcolo della MSD tra due utenti. Formalmente definiamo la nuova metrica nel seguente modo:

$$\forall x, y \in Users \quad newMetric(x, y) = Jaccard(x, y) \times (1 - MSD(x, y))$$

Se la URM è normalizzata nell'intervallo $[0; 1]$ allora ciascuno dei fattori di cui sopra è nell'intervallo $[0; 1]$

Esempio:

$$r_x = (4\ 5\ missing\ 3\ 2\ missing\ 1\ 1) = (0.75\ 1\ missing\ 0.5\ 0.25\ missing\ 0\ 0)$$
$$r_y = (4\ 3\ 1\ 2\ missing\ 3\ 4\ missing) = (0.75\ 0.5\ 0\ 0.25\ missing\ 0.5\ 0.75\ missing)$$

allora

$$d_{x,y} = (0\ 0.25\ missing\ 0.063\ missing\ missing\ 0.563)$$
$$MSD(x, y) = (0 + 0.25 + 0.063 + 0.563) / 4 = 0.218$$
$$Jaccard(x, y) = 4 / (6 + 6 - 4) = 0.5$$

$$\therefore\ newMetric(x, y) = 0.5 \times (1 - 0.218) = 0.391$$

2.5 Predire un voto

Per stimare il voto che un utente *x* darebbe ad un film *i* che non ha votato, si può procedere nel seguente modo:

1. Determino K_x attraverso l'algoritmo k-nearest neighbors ottenendo i *k* utenti più simili a x
2. Tra questi seleziono solo quelli che hanno espresso un voto significativo (non missing) per il film in questione. Formalmente $G_{x,i} = \{n \in K_x : \exists r_{n,i} \neq missing\}$
3. Se $G_{x,i} \neq \emptyset$, allora applico una funzione di aggregazione sugli elementi di $G_{x,i}$

Assumendo che l'insieme $G_{x,i}$ sia non vuoto, le principali funzioni di aggregazione che vengono usate nel contesto dei RS[AGG] sono:

- La media (average aggregation): $p_{x,i} = \frac{1}{|G_{x,i}|} \sum_{n \in G_{x,i}} r_{n,i}$
- La somma pesata (weighted sum aggregation): $p_{x,i} = \mu_{x,i} \sum_{n \in G_{x,i}} sim(x, n)\, r_{n,i}$

- La somma pesata corretta rispetto alla media (adjusted weighted sum/deviation-from-mean):

$$p_{x,i} = \overline{r_x} + \mu_{x,i} \sum_{n \in G_{x,i}} sim(x,n)(r_{n,i} - \overline{r_n})$$

dove $\mu = 1 / \sum_{n \in G_{x,i}} sim(x,n)$ rappresenta un fattore di normalizzazione (normalizing factor).

3. Implementazione

In questo capitolo verranno riportati e descritti i principali algoritmi implementati.

3.1 Panoramica sul progetto

Il sistema che si intende realizzare sarà composto da quattro moduli principali che dovranno rispettivamente occuparsi del caricamento dei dati, la valutazione delle metriche, l'implementazione dei metodi di aggregazione e la valutazione delle performance.

3.1.1 Linguaggio e ambiente di sviluppo

Il sistema di raccomandazione è stato implementato usando il linguaggio Julia[JU] nella sua versione 1.8.5 utilizzando un sistema basato su Linux versione 5.19.0 e uno basato su MacOS versione 13.2, mentre i tempi di predizione sono stati raccolti utilizzando un sistema Linux 5.19.0 con CPU *AMD Ryzen 3700X* e 16GiB RAM 3600MT/s CL18, utilizzando 8 thread.

3.1.2 Dipendenze

Oltre al runtime di Julia è necessario installare, attraverso il gestore dei pacchetti del linguaggio, le seguenti dipendenze:

- `Statistics`: necessario per le principali formule statistiche, quali media e deviazione standard
- `StasBase`: necessario per il calcolo della moda campionaria
- `Plots`: necessario per poter stampare a schermo i grafici
- `CSV`: necessario per leggere il dataset
- `DataFrames`: utilizzato come struttura dati per la lettura del dataset

3.1.3 Struttura dei file

Di seguito è riportata la struttura dei file che compongono il programma:

- ***aggregation_methods.jl***: contiene il codice necessario per implementare i tre metodi di aggregazione ed il fattore di normalizzazione μ ad essi correlato
- ***analytics.jl***: contiene il codice usato per generare i report analitici sulla distribuzione del dataset
- ***dataset_injection.jl***: contiene quanto necessario per caricare il dataset
- ***dataset_split.jl***: contiene il codice necessario per dividere un dataset in due sottoinsiemi, utilizzato per separare test e validation set
- ***main.jl***: il punto di ingresso dell'applicazione. È dove vengono definiti gli iperparametri.
- ***metrics.jl***: implementa le metriche di similarità precedentemente descritte

- ***performance_evaluation.jl***: contiene il codice necessario per fare previsioni e valutare le performance del RS
- ***rs.jl***: contiene il codice per costruire la URM e per individuare i k-nearest neighbors di un utente

3.2 Caricamento dei dati

All'avvio del programma viene caricato il dataset in memoria chiamando la seguente funzione

```
function loadData(dataset)
   downloadDataset(dataset)

   println("Loading $dataset dataset...")
   moviesDataFrame = DataFrame(CSV.File(dataset * "/movies.csv"))
   ratingsDataFrame = DataFrame(CSV.File(dataset * "/ratings.csv"))

   println("✓ Dataset loaded\n")
   return moviesDataFrame, ratingsDataFrame
end
```

Attraverso l'invocazione al metodo `downloadDataset` il dataset viene scaricato su disco in formato .zip ed estratto invocando il comando shell `unzip`. Se questo non fosse disponibile, sarà necessario estrarre manualmente l'archivio.

Una volta scaricato, il dataset viene caricato in memoria usando una struttura dati tabellare nota come `DataFrame`.

Attraverso la funzione `datasetSplit`, l'insieme $\{(i,j) : r_{ij} \neq missing\}$ viene partizionato in training set e test set.

```
function datasetSplit(ratingsDataFrame, foldSize, foldIndex)
   numberOfRatings = size(ratingsDataFrame, 1)
   numberOfTestSetRatings = floor(Int, numberOfRatings * foldSize)

   testSetStartIndex = foldIndex * numberOfTestSetRatings + 1 # +1 because Julia's
indices start from 1
   testSetEndIndex = testSetStartIndex + numberOfTestSetRatings - 1 # -1 because
Julia intervals are inclusive

   indices = range(1, numberOfRatings)
   foldIndices = indices[testSetStartIndex:testSetEndIndex]
   trainingSetIndices = filter(x -> x ∉ foldIndices, indices)

   foldDataFrame = ratingsDataFrame[ foldIndices, :]
   trainingSetDataFrame = ratingsDataFrame[ trainingSetIndices, :]

   return trainingSetDataFrame, foldDataFrame
```

```
end
```

Questa funzione prende in ingresso il DataFrame contenente i rating da dividere, un numero a virgola mobile foldSize che indica che percentuale di rating da riservare per il foldDataFrame in output e un intero foldIndex che indica da quale posizione del dataset cominciare ad estrarre il foldDataFrame.

Fig 3.2.1 : *esempio di divisione con foldSize = 0.1 e foldIndex = 0*

Fig 3.2.2 : *esempio di divisione con foldSize = 0.1 e foldIndex = 1*

In figura 3.2.1 e 3.2.2 vengono riportati degli esempi di suddivisione del dataset. In rosso viene indicata la porzione riservata per il foldDataFrame, mentre la restante parte viene restituita in trainingSetDataFrame.

3.3 Costruzione della URM

Una volta caricati i dati e opportunamente divisi in training e test set, viene costruita la User Rating Matrix (URM) attraverso la seguente funzione:

```
function buildURM(ratingsDataFrame, numberOfUsers, numberOfMovies,
normalizeURM::Bool=true)
   URM = allocateMatrix(numberOfUsers, numberOfMovies)
   @threads for i=1:numberOfUsers
       userId = i
       userRatings = getUserRatings(ratingsDataFrame, userId)
       for j=1:numberOfMovies
           movieId = getMovieId(moviesDataFrame, j)
           URM[i,j] = getUserRatingByMovieId(userRatings, movieId)
       end
   end
   if normalizeURM
       # rating range: [0.5; 5]
       URM = normalize(URM, getRatingRange())  # URM: [0; 1]
   end
   return URM
end
```

La funzione prende in ingresso la tabella dei ratings sotto forma di DataFrame e il numero di utenti e di film, oltre a un parametro booleano opzionale che, se specificato, induce la normalizzazione della URM, portando i voti da un generico range (da 0.5 a 5 nel caso di MovieLens) in numeri a virgola mobile compresi tra 0 e 1

La funzione provvede ad allocare una matrice di dimensioni opportune e per ogni utente estrae il suo vettore dei voti, per ognuno di questi, riempie le caselle della URM. Nelle posizioni non riempite ci sarà il valore `missing` di Julia.

Infine questa funzione usa la keyword `@threads` in corrispondenza del ciclo for per indicare al runtime Julia di eseguire il ciclo in modo multithreaded nell'ottica di ridurre i tempi d'esecuzione.

3.4 Implementazione delle metriche

Nel file metrics.jl vengono implementate sia la correlazione di Pearson che la NewMetric presentata nel paper di riferimento[PAP]. Entrambe le metriche prendono in ingresso due vettori che rappresentano i rating degli utenti di cui si vuole determinare la similarità.

3.4.1 Correlazione di Pearson

```
function pearsonCorrelation(x, y)
    x, y = commonRatings(x, y)

    if length(x) == 0
        return missing
    end

    return cor(x, y)
end
```

La funzione prende in ingresso i due vettori e, mediante chiamata a `commonRatings`, li filtra in modo che contengano solo i rating legati ad item votati da entrambi gli utenti, dopodiché chiama la funzione `cor` del pacchetto Statistics per ottenere la correlazione. Siccome questa non è definita per insiemi vuoti, viene prima fatto un controllo sulla lunghezza dei vettori e, se questi dovessero risultare vuoti (non ci sono rating in comune) viene restituito il valore missing, ad indicare che il sistema non è in grado di fornire una misura di similarità tra i due utenti causa dati insufficienti.

3.4.2 New Metric

```
function newMetric(x, y)
    squaredDifferences = squaredDifference(x, y)
    nonMissingSquaredDifferences = collect(skipmissing(squaredDifferences))
    lengthOfNonMissingSquaredDifferences = length(nonMissingSquaredDifferences)

    if (lengthOfNonMissingSquaredDifferences == 0)
        return missing
    end
```

```
    jaccardValue = jaccard(x, y, lengthOfNonMissingSquaredDifferences)
    msd = mean(nonMissingSquaredDifferences)
    return jaccardValue * (1 - msd)
end
```

La funzione newMetric prende in ingresso i due vettori di rating degli utenti che si intende confrontare e restituisce una misura di similarità nell'intervallo [0, 1].
Viene dapprima calcolato il vettore delle differenze al quadrato tra i rating di pari posto. Tale vettore conterrà valori missing in corrispondenza degli item per cui almeno uno dei due utenti non ha espresso alcun giudizio. Questi valori vengono scartati tramite l'istruzione `collect(skipmissing(squaredDifferences))`. Infatti `skipmissing` fornisce un iteratore in grado di saltare eventuali valori missing, mentre `collect` colleziona tali valori materializzandoli in un array. Analogamente alla misura di Pearson, nel caso in cui gli utenti non hanno alcun rating in comune viene restituito il valore missing. Alternativamente si procede al calcolo dello Jaccard e, infine, si combinano Jaccard e Mean Squared Difference per fornire la misura di similarità.

L'indice di Jaccard è così implementato:

```
function jaccard(x, y, lengthOfNonMissingSquaredDifferences)
    numberOfNonMissingValuesInX = length(collect(skipmissing(x)))
    numberOfNonMissingValuesInY = length(collect(skipmissing(y)))

    return lengthOfNonMissingSquaredDifferences / (numberOfNonMissingValuesInX +
numberOfNonMissingValuesInY - lengthOfNonMissingSquaredDifferences)
end
```

Si contano il numero di rating non-missing forniti da x e il numero di rating non-missing fornito da y. Viene poi restituito il rapporto tra il numero di elementi votati da entrambi rispetto al numero di elementi diversi votati in totale dai due utenti.

3.5 Implementazione dei metodi di aggregazione

I metodi di aggregazione sono le funzioni che permettono, fornito un utente e un item, di fornire una previsione di rating per quell'utente e quell'item, basandosi sulla URM.

Nel file aggregation_methods.jl vengono implementati i tre metodi di aggregazione presentati nel Paragrafo 2.5 e il relativo fattore di normalizzazione. Tutti e tre i metodi fanno uso dell'insieme $G_{x,i}$ contenente gli *n* utenti più vicini (in termini di similarità) all'utente *x* dato, che abbiano espresso un voto per l'item *i* di cui si vuole fare previsione.

3.5.1 K Nearest Neighbors

```
function kNearestNeighbors(urm, userRatings, k, metric=newMetric)
    numberOfUsers = size(urm, 1)
```

```
    # Allocate the similarity matrix, having all the users as rows and
    # storing the userId in the first column and the similarity in the second column
    userSimilaritiesMatrix = allocateMatrix(numberOfUsers, 2)
    userIdColumn = 1
    similarityColumn = 2


    # For each user in urm, append similarity to similarities
    for i=1:numberOfUsers
        similarity = metric(userRatings, urm[i, :])
        userSimilaritiesMatrix[i, userIdColumn] = i
        userSimilaritiesMatrix[i, similarityColumn] = similarity
    end

    # Sort similarities by the second column (similarity) descending order
    similarities = userSimilaritiesMatrix[sortperm( -userSimilaritiesMatrix[:,
similarityColumn]), :]

    # Retrieve the first k indices (userIds)
    # if k > numberOfUsers, this prevents returning more items than there actually
are
      in the matrix
    n = min(k, numberOfUsers)
    return Int.(similarities[1:n, userIdColumn]) # returns the userIds (casted to
Integers) ordered by similarity DESC
end
```

Alla base di tutti i metodi di aggregazione c'è questo algoritmo. Data in input una URM, i rating di un utente *user*, un valore *k* e una metrica di similarità, fornisce i primi *k* utenti estratti dalla URM più vicini ad *user* in base alla metrica di similarità scelta.

In particolare viene allocata una matrice con tante righe quanti sono gli utenti nella URM e due colonne: una per memorizzare l'identificativo del vicino *i*-esimo e l'altra per memorizzare la sua similarità rispetto all'utente in input. Tale matrice viene riempita iterando su tutti gli utenti del dataset e infine viene ordinata per similarità in ordine decrescente. Il risultato sarà una matrice contenente coppie (userId, similarità). Di queste ne vengono selezionate le prime *k*, se *k* < *numberOfUsers*, altrimenti vengono selezionate tutte le righe. Infine vengono restituiti gli userId di cui viene fatto un cast a numero intero.

3.5.2 knnWhichRatedItem

Una volta ottenuti i *k* utenti più vicini all'utente specificato, questi vengono filtrati scartando tutti quelli che non hanno votato l'item di interesse, ottenendo così l'insieme $G_{x,i}$.

```
function knnWhichRatedItem(URM, userRatings, k, itemIndex, metric=newMetric)
   # Get user's k-nearest neighbors
   knn = kNearestNeighbors(URM, userRatings, k, metric)
   knnWhichRatedItem = []

   # For each k-nearest-neighbor: if neighbor has rated item then add to G_u,i
   for n in eachindex(knn)
       neighborUserId = knn[n]
       if !ismissing(URM[neighborUserId, itemIndex])
           append!(knnWhichRatedItem, neighborUserId)
       end
   end

   # userIds
   return knnWhichRatedItem
end
```

Questa funzione ottiene i k utenti più vicini ad *userRatings* mediante chiamata a `kNearestNeighbors`, dopodiché costruisce una lista in cui inserisce i soli utenti più vicini che però hanno espresso un voto per l'item indicato. Infine restituisce tale lista in output.

3.5.3 Average Aggregation

Il più semplice tra i metodi di aggregazione, fornisce come valore di previsione la media aritmetica dei rating forniti dai *k* utenti più vicini all'utente dato. In caso in cui non esista alcun utente vicino all'utente dato che abbia espresso un rating per l'item indicato, viene restituito il valore `missing`.

```
function averageAggregation(urm, userRatings, itemIndex, k, metric=newMetric)
   knnForItem = knnWhichRatedItem(urm, userRatings, k, itemIndex, metric)  # G_u,i

   if isempty(knnForItem)
       return missing
   end

   neighborsRatings = urm[knnForItem, itemIndex]  # r_n,i
   return mean(neighborsRatings)
end
```

Siccome `knnForItem` è una lista contenente gli indici del vicinato significativo per l'utente, allora accedendo alla URM fornendo tale lista sulla prima dimensione e il valore *itemId* sulla seconda dimensione, si ottengono i voti che i vicini hanno espresso per quell'item. Infine viene restituita la loro media.

3.5.4 Weighted Sum Aggregation

Questa seconda metrica fornisce in output un valore di previsione calcolato come la media dei rating forniti dagli utenti vicini all'utente dato, pesata per il loro valore di similitudine. Il tutto viene infine moltiplicato per un fattore di normalizzazione μ.

```
function weightedSumAggregation(urm, userRatings, itemIndex, k, metric=newMetric)
   knnForItem = knnWhichRatedItem(urm, userRatings, k, itemIndex, metric)  # G_u,i

   if isempty(knnForItem)
       return missing
   end

   sum = 0.0
   similarities_sum = 0.0

   for neighborId in eachindex(knnForItem)
       neighborRatings = urm[knnForItem[neighborId], :]
       neighborRating = neighborRatings[itemIndex]  # r_n,i

       similarity = metric(userRatings, neighborRatings)
       similarities_sum = similarities_sum + similarity
       sum += similarity * neighborRating
   end

   if similarities_sum == 0
       return missing
   end

   mi = 1 / similarities_sum
   return sum * mi
end
```

Come nel caso precedente vengono raccolti i k utenti più vicini all'utente dato e di questi vengono considerati solo quelli che hanno espresso un voto per l'item di interesse. Per ognuno di questi vicini viene calcolata la similarità e si pesa il voto per tale similarità. Infine viene restituita in output la somma pesata dei rating moltiplicato per un fattore di normalizzazione μ pari al reciproco della somma delle similarità. Se la somma delle similarità risulta nulla (cioè gli utenti più vicini all'utente dato che hanno anche votato l'item in questione hanno tutti similarità nulla) allora la funzione restituisce `missing`.

3.5.5 Adjusted Weighted Sum

Questa terza e ultima metrica è un'evoluzione della precedente che tiene traccia del voto medio dell'utente di cui si vuole fare inferenza e degli utenti ad esso vicini.

```
function adjustedWeightedSumAggregation(urm, userRatings, itemIndex, k,
metric=newMetric)
    knnForItem = knnWhichRatedItem(urm, userRatings, k, itemIndex, metric)  # G_u,i

    if isempty(knnForItem)
        return missing
    end

    sum = 0.0
    similarities_sum = 0.0

    for neighborId in eachindex(knnForItem)
        neighborRatings = urm[knnForItem[neighborId], :]
        neighborRating = neighborRatings[itemIndex]  # r_n,i
        neighborAvgRating = mean(collect(skipmissing(neighborRatings)))

        similarity = metric(userRatings, neighborRatings)
        similarities_sum = similarities_sum + similarity
        sum += similarity * (neighborRating - neighborAvgRating)
    end

    if similarities_sum == 0
        return missing
    end

    userAvgRating = mean(collect(skipmissing(userRatings)))
    mi = 1 / similarities_sum
    return userAvgRating + sum * mi
end
```

L'implementazione è simile al caso precedente, ma il voto degli utenti vicini è preso rispetto alla media dei loro voti, in modo da eliminare l'influenza di eventuali bias. In più il risultato in output è sommato alla media dei voti dell'utente stesso, in modo da introdurre nel risultato dell'inferenza il bias dell'utente e proporre una votazione più vicina a quella che darebbe l'utente stesso.

3.6 Valutazione delle performance

La valutazione delle prestazioni è eseguita dalla funzione `computePredictions`, così implementata:

```
function computePredictions(trainingURM, targetDataFrame, targetURM,
aggregationMethod, k, metric)
   testSetItemCount = size(targetDataFrame, 1)

   # Extract and normalize target ratings
   targets = targetDataFrame[:, :rating]
   # rating range: [0.5; 5]
   targets = normalize(targets, getRatingRange())

   # Allocate predictions vector
   predictions = Vector{Union{Missing, Float64}}(undef, testSetItemCount)

   # We keep track of how long the predictions took. The variable is atomic because
the following for loop is multithreaded
   totalTime = Threads.Atomic{Float64}(0)

   # We keep track of how many predictions we computed so far to give a progress
report
   counter = Threads.Atomic{Int64}(0) # counter is an atomic variable of type Int64
initialized to 0

   # for each test set item
   @threads for i = 1:testSetItemCount

       # Print progress
       printErrorProgress(counter, testSetItemCount)

       # Extract relevant information of the current prediction
       userId = targetDataFrame[i, :userId]
       movieId = targetDataFrame[i, :movieId]
       itemIndex = getMovieIndexById(movieId)

       user = targetURM[userId, :]

       timeBefore = time() # time, in seconds, before the prediction started
       predictions[i] = aggregationMethod(trainingURM, user, itemIndex, k, metric)
# compute prediction

       # We now calculate how long the prediction took and add it to the totalTime
atomic variable
       elapsedTime = time() - timeBefore
       Threads.atomic_add!(totalTime, elapsedTime)
   end

   printTimeReport(totalTime, testSetItemCount)

   return targets, predictions
end
```

La funzione prende in ingresso la URM che rappresenta il ‘modello’ corrente, il DataFrame contenente i rating di test e la URM di test. Inoltre riceve in ingresso i parametri necessari per la singola inferenza, quali il metodo di aggregazione, la dimensione del vicinato e la metrica di similarità.

La funzione estrae e normalizza tra 0 e 1 i rating dal DataFrame target e alloca un vettore di predizioni. Per ogni rating di test (in modo parallelo) estrae userId, itemIndex e il vettore di rating dell’utente e chiama il metodo di aggregazione. Viene calcolato il tempo impiegato per eseguire il metodo di aggregazione e si somma ad un contatore atomico. Viene inoltre stampato un report di progresso che utilizza un contatore atomico per tenere traccia della percentuale di avanzamento. Alla fine di ogni inferenza viene effettuata una chiamata esplicita al garbage collector di Julia per evitare problemi di memory leak.

Infine vengono restituiti i vettori target e predictions, contenenti rispettivamente le golden label e i risultati delle inferenze. Tali vettori saranno poi utilizzati per chiamare la funzione di errore, calcolare precision e racall o il numero di predizioni perfette.

4. Analisi sperimentale

In questo capitolo verranno presentati i risultati sperimentali ottenuti al variare degli iperparametri (metodo di aggregazione e dimensione del vicinato). Ad ogni iterazione della k-fold CV si ottiene una tupla di prestazioni sul test set (MAE, precision, recall, f-measure e numero di predizioni perfette). Alla fine di tutte le iterazioni, la valutazione finale del sistema viene calcolata come la media delle prestazioni ottenute ad ogni iterazione, tenendo in considerazione anche la deviazione standard.

Alla fine della procedura è stato osservato che le deviazioni standard sono confrontabili ed è stato scelto il predittore che minimizza la MAE.

Il dataset è stato partizionato in 90% di dati per il training e il 10% di dati per il testing. La dimensione del vicinato è stata fatta variare tra 1 e 150 a passi di 5. Queste quantità sono state scelte osservando che sul paper di riferimento[PAP] tale quantità viene fatta variare tra 2 e 1500 a passi di 50 utilizzando un dataset 10 volte più grande di quello utilizzato in questo lavoro. Per contenere i tempi d’esecuzione è stata utilizzata la 5-fold cross validation siccome $k \in \{5, 10\}$ sono le scelte più comunemente fatte in letteratura.

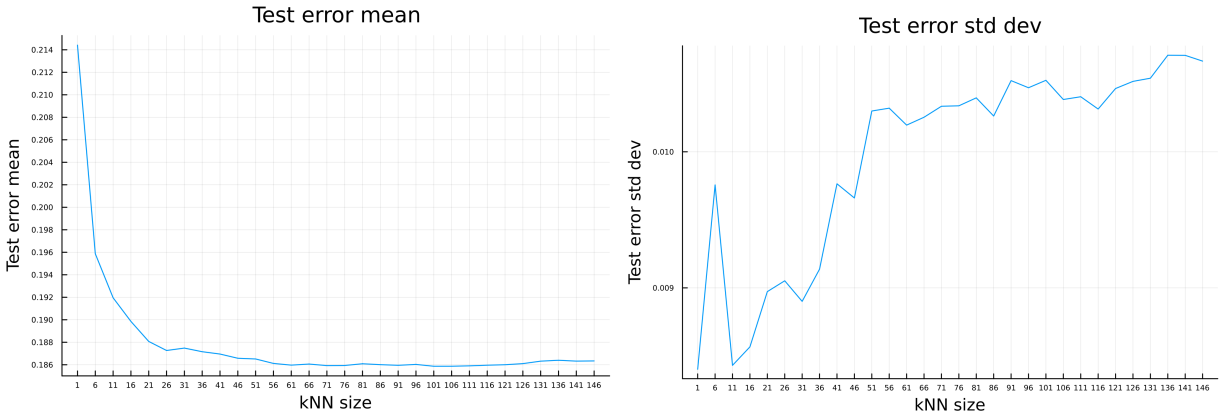
Il numero di predizioni perfette è calcolato come il numero di predizioni che non distano più di mezzo voto dalla golden label/ground truth.

Gli iperparametri scelti sono:

- Validation technique: 5-fold cross validation

- Total data is split into 90.0% for training and 10.0% for testing
- Similarity metric: newMetric
- Aggregation function in {Average, Weighted Sum, Adjusted Weighted Sum}
- Error function: meanAbsoluteError
- Neighborhood size: [1:150]
- Neighborhood step: 5

4.1 Average aggregation



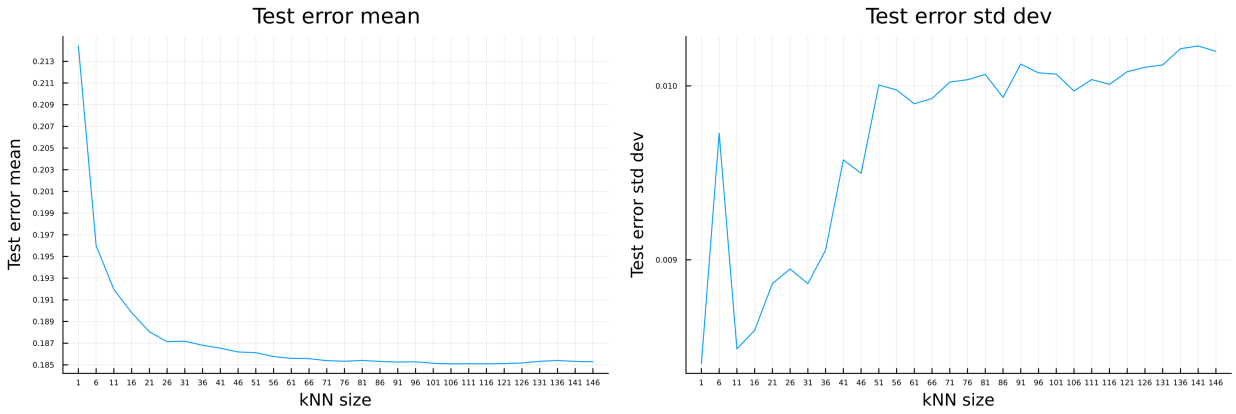
Scegliendo l'average come funzione di aggregazione, il modello che ha ottenuto le prestazioni migliori in termini di MAE è quello con dimensione del vicinato k pari a 101, cui corrisponde una MAE di 0.181 e che ha prodotto il seguente risultato:

- Mean Absolute Error (MAE): 0.186 $\pm$ 0.011
- Precision: 0.739
- Recall: 0.730
- F-measure: 0.733
- Numero di predizioni perfette: 3869.8/10083 (38.38%)

Il tempo di esecuzione medio per una singola predizione è stato di 0.077s

La MAE ha un andamento simile alla funzione razionale $f(x) = 1/x$. Tale errore diminuisce con la massima rapidità quando la grandezza del vicinato passa da 1 a 6. Poi tende a decrementare in maniera più graduale quando la grandezza del vicinato passa da 6 a 81 per poi assestarsi intorno ad un valore costante pari a 0.181. In particolare a partire da k= 106, l'errore sul test set sembra gonfiarsi nuovamente con l'aumentare del vicinato, ma restando comunque su valori bassi. La deviazione standard raggiunge il suo minimo per k=11, dopodiché aumenta col crescere della dimensione del vicinato. I valori sono compresi tra 0.008 e 0.011.

4.2 Weighted sum aggregation



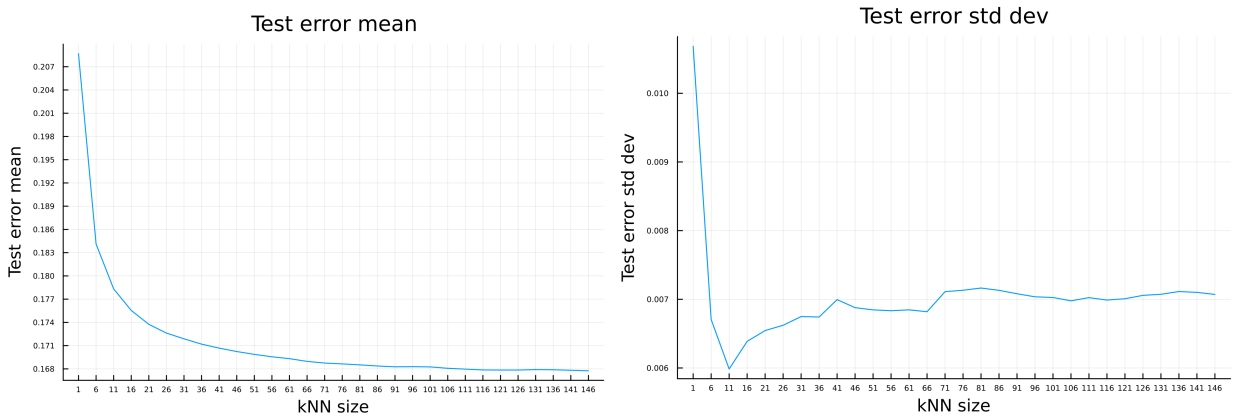
Scegliendo la weighted sum come funzione di aggregazione, il modello che ha ottenuto le prestazioni migliori in termini di MAE è quello con dimensione del vicinato k pari a 106, cui corrisponde una MAE di 0.185 e che ha prodotto il seguente risultato:

- Mean Absolute Error (MAE): 0.185 $\pm$ 0.010
- Precision: 0.745
- Recall: 0.720
- F-measure: 0.731
- Numero di predizioni perfette: 4062.2/10083 (40.29%)

Il tempo di esecuzione medio per una singola predizione è stato di 0.076s

Similmente alla funzione di aggregazione precedente, possiamo notare che all'aumentare della dimensione del vicinato la MAE decresce come una funzione razionale del tipo $f(x) = 1/x$ fino ad assestarsi attorno ad un valore. Differentemente da prima, però, a partire da k=121 la funzione non si gonfia ma continua a decrescere fino a raggiungere il suo minimo per k=106. Notando che utilizzando la average aggregation la MAE ha valore minimo 0.186 per k=101, si conclude che la weighted sum aggregation non porta alcun miglioramento in termini di errore.

4.3 Adjusted weighted sum aggregation



Infine, scegliendo la adjusted weighted sum come funzione di aggregazione, il modello che ha ottenuto le prestazioni migliori in termini di MAE è quello con dimensione del vicinato k pari a 146 e che ha prodotto il seguente risultato:

- Mean Absolute Error (MAE): 0.168 $\pm$ 0.007
- Precision: 0.793
- Recall: 0.727
- F-measure: 0.758
- Numero di predizioni perfette: 4632.4/10083 (45.94%)

Il tempo di esecuzione medio per una singola predizione è stato di 0.087s

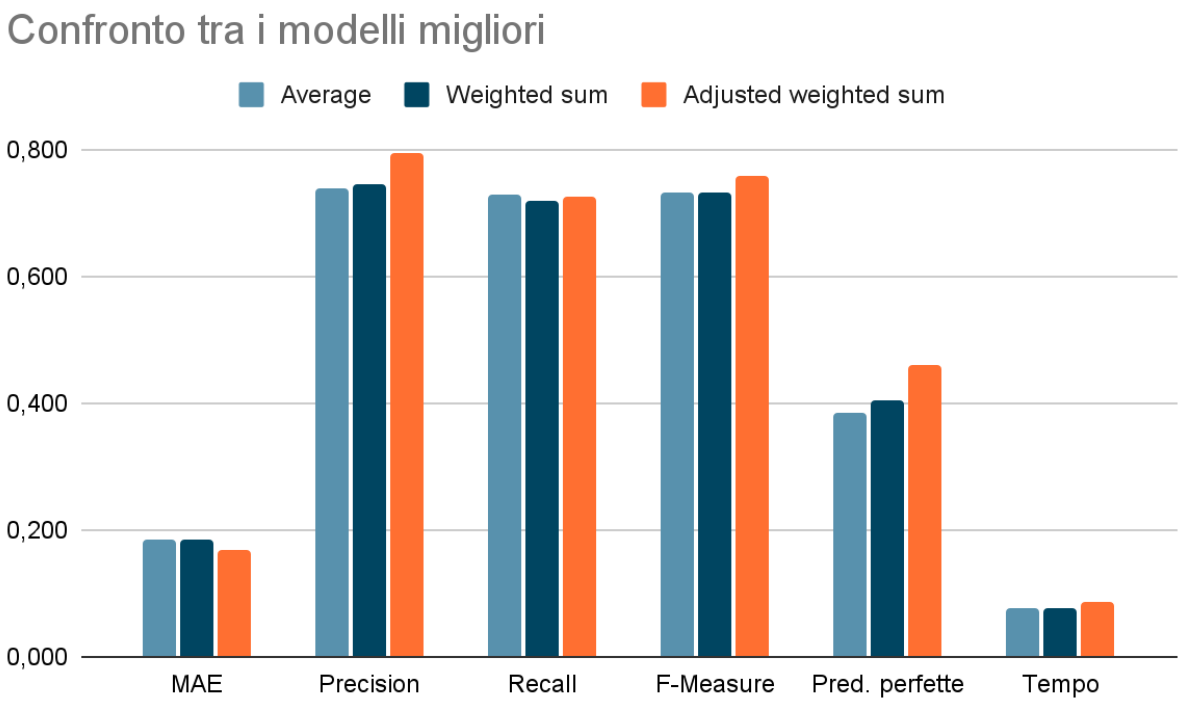
Ancora una volta la MAE sembra decrescere come la funzione razionale $f(x) = 1/x$. Si può individuare il suo minimo per k=146 che di fatto è il massimo valore possibile per come abbiamo scelto lo span del vicinato. Questo suggerisce, in maniera del tutto empirica, che aumentando tale span, potrebbe capitare che la funzione continui a decrescere individuando un altro punto di minimo k. Anche la deviazione standard risulta essere “migliore” rispetto ai due casi precedenti, siccome più piccola e schiacciata.

4.4 Confronto dei metodi di aggregazione

La seguente tabella sintetizza l’analisi sperimentale di cui sopra. In grassetto sono evidenziati i migliori risultati per ciascun modello. Siccome il vicinato di un utente *x* è un sottoinsieme stretto degli utenti del RS, un vicinato più piccolo – a parità di prestazioni – è probabilmente da preferire, in quanto il sistema sarebbe in grado di fornire una raccomandazione più o meno accurata anche in quei dataset con un ridotto numero di utenti. In generale, per valori di k troppo piccoli ci sono pochi dati per una stima affidabile; per valori troppo grandi, invece, viene

introdotto del rumore (data noise).

	kNN	MAE	Precision	Recall	F-Measure	Pred. perfette	Tempo(s)
Average	101	0,186	0,739	**0,730**	0,733	38,38%	**0,077**
Weighted sum	106	0,185	0,745	0,720	0,731	40,29%	0,076
Adjusted weighted sum	146	**0,168**	**0,793**	0,727	**0,758**	**45,94%**	0,087



Come si osserva non c'è differenza significativa, in termini di MAE, tra il miglior modello basato su average aggregation e il miglior modello basato su weighted sum aggregation. Inoltre risultano pressoché equivalenti in termini di precision e recall; d'altro canto, sebbene il modello che sfrutta la weighted sum aggregation risulti essere migliore in termini di predizioni perfette, impiega quasi il doppio del tempo. Per quanto riguarda il tempo di esecuzione, il modello basato su weighted sum risulta visibilmente peggiore del modello basato su adjusted weighted sum; più precisamente lo stesso risulta il peggiore in tutte le altre metriche, tranne che in termini di predizioni perfette (dove il modello che sfrutta average risulta ancora peggiore): se ne deduce, quindi, che la weighted sum risulta un cattivo compromesso tra gli altri due e si dovrebbe scegliere, in base alle necessità, soltanto tra average aggregation e adjusted weighted sum aggregation.

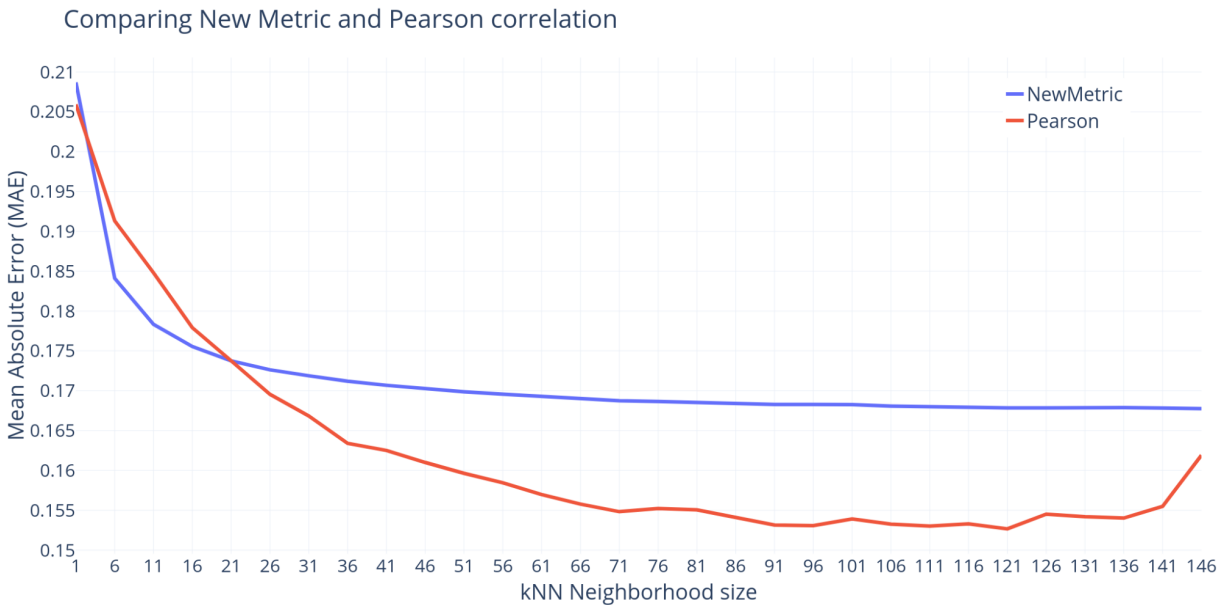
Il modello che ha ottenuto i risultati migliori in termini di MAE, precision, recall, F-measure e numero di predizioni perfette è quello che sfrutta la adjusted weighted sum come metodo di aggregazione. Confrontandolo col modello basato su average aggregation, inoltre, notiamo che a

fronte di una diminuzione relativa della MAE del 9.68% il tempo di calcolo aumenta del 13%. Si osserva la capacità dell'adjusted weighted sum di scalare meglio col numero di utenti considerati nel vicinato. Notiamo inoltre come questo metodo sia in grado di fornire delle predizioni più accurate per qualsiasi dimensione del vicinato.

Come già notato, l'adjusted weighted sum riesce ad ottenere con una dimensione del vicinato $k = 6$ le stesse prestazioni che l'average aggregation riesce ad ottenere con $k = 101$.

4.5 Confronto con Pearson

La seguente Figura mostra l'andamento della MAE al variare della grandezza del vicinato k utilizzando, come metrica di similarità, NewMetric (in blu) e l'indice di correlazione di Pearson (in arancione). Come si nota, NewMetric funziona particolarmente bene per $6 \leq k < 21$. Quando $k = 21$ le due funzioni si intersecano e per $k > 21$ l'indice di Pearson sembra funzionare meglio; ma l'incremento repentino della curva arancione osservato per $k \geq 141$ ci fa pensare che, a dimensioni del vicinato molto alte, Pearson possa perdere di qualità e la NewMetric possa comportarsi meglio. Questo fenomeno potrebbe essere dovuto al fatto che, in accordo alle osservazioni dell'articolo scientifico di riferimento[PAP], i miglioramenti dovuti alla nuova metrica di similarità sono apprezzabili quando lo span di voti esprimibili non è troppo esteso: le vecchie versioni di MovieLens, ammettono uno span di voti esprimibili da 1 a 5 mentre le nuovi versioni del dataset ammettono voti da 1 a 5 e i mezzi voti (10 voti esprimibili in totale).

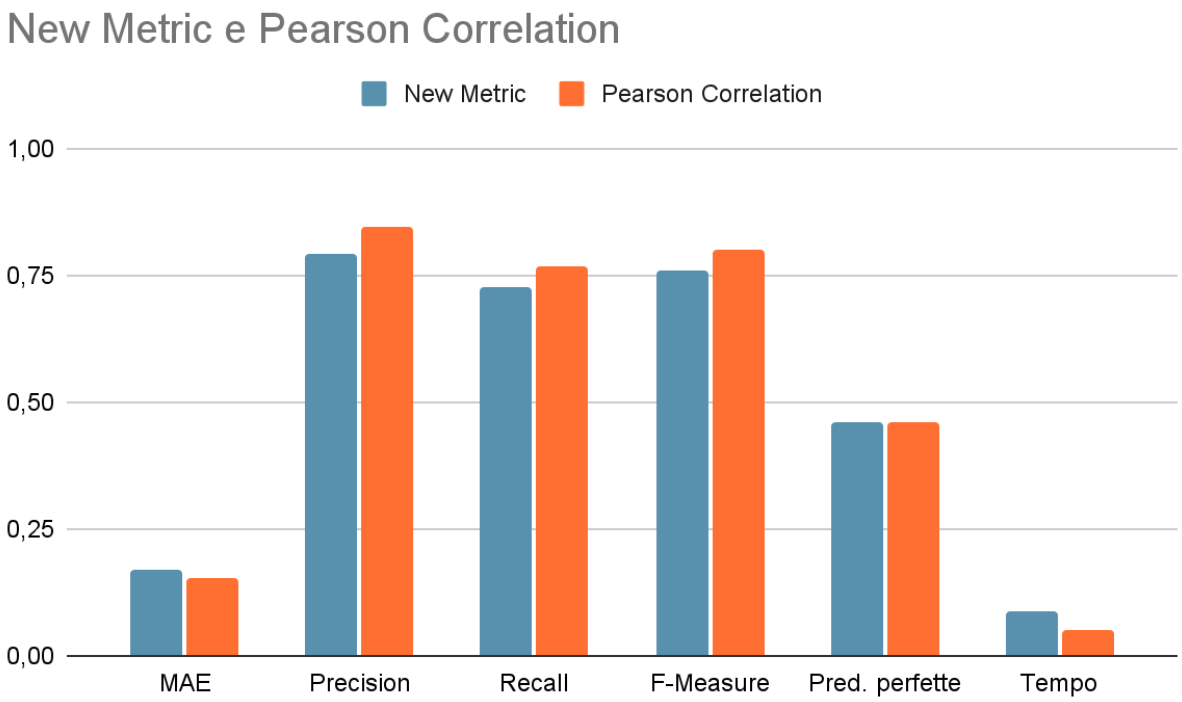


L'andamento non monotòno della funzione di errore associata all'indice di Pearson fa sì che la scelta di una dimensione del vicinato *k* grande sia garanzia di successo per la NewMetric, ma

non per l'indice di Pearson, per il quale è necessaria un'analisi sperimentale del problema per individuare il valore di *k* che garantisce il risultato migliore.

La seguente tabella sintetizza le prestazioni dei migliori modelli utilizzando New Metric oppure l'indice di correlazione di Pearson.

	kNN	MAE	Precision	Recall	F-Measure	Pred. perfette	Tempo (s)
New Metric	146	0,168	0,793	0,727	0,758	45,94%	0,087
Pearson Correlation	121	**0,153**	**0,844**	**0,766**	**0,802**	**45,97%**	**0,050**



Notiamo che l'indice di Pearson offre una diminuzione relativa dell'errore di circa il 10% impiegando il 57% del tempo, garantendo al tempo stesso risultati marginalmente migliori nelle altre metriche.

5. Sviluppi futuri

Una volta che la migliore configurazione di iperparametri è stata validata, si può pensare di fare un dump del recommender system per un eventuale deploy. Per quanto riguarda la parte di ottimizzazione e re-engineering del sistema, si potrà pensare di utilizzare altre strutture dati per implementare il vicinato e di memorizzare in cache il valore di similarità per evitare il ricalcolo durante l'inferenza. O magari attrezzarsi di macchine più potenti per aumentare il numero di fold da 5 a 10 e aumentare lo span del vicinato.

6. Conclusioni

Nei sistemi di raccomandazione sembra che gli utenti tendano a recensire un item in base ad una pseudo-classificazione "Mi è piaciuto"/"Non mi è piaciuto", per poi trasferire tutto ciò sotto forma di voto. Questo fenomeno si riflette, come visto nel Capitolo 2, in piccole variazioni intorno al voto medio. In particolare nei dataset caratterizzati da uno span di voti particolarmente piccolo, come accade nelle vecchie versioni di MovieLens oppure su NetFlix in cui i voti esprimibili sono soltanto 5, si è notato che le informazioni non-numeriche date dall'indice di Jaccard non possono bastare perché indebolirebbero il processo di predizione, rendendo necessario complementare informazioni di questo tipo con quelle puramente numeriche. Sebbene l'introduzione di una nuova metrica basata su Jaccard e MSD migliori accuratezza, precision, recall e numero di predizioni perfette su un dataset come il vecchio MovieLens o Netflix – in cui lo span dei voti esprimibili è sufficientemente piccolo – la stessa combinazione non porta invece alcun miglioramento sostanziale quando è applicata ad un dataset che permette voti in un intervallo più ampio, ad esempio tra 1 e 10 come accade per FilmAffinity oppure per le nuove versioni di MovieLens. In questi casi ha senso continuare ad usare l'indice di Pearson perché la nuova metrica non migliora le sue prestazioni - nonostante i limiti teorici. L'intervallo dei voti, che può sembrare in un primo momento una scelta trascurabile, risulta invece piuttosto delicata.

Bibliografia

[PAP] BOBADILLA, Jesús; SERRADILLA, Francisco; BERNAL, Jesus. A new collaborative filtering metric that improves the behavior of recommender systems. *Knowledge-Based Systems*, 2010, 23.6: 520-528; 357 citations; Q1 on scimagojr

[DS] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872

[JU] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *Society for Industrial and Applied Mathematics*, 2017, 10.1137/141000671

[MET] J.L. Sanchez, F. Serradilla, E. Martinez, J. Bobadilla, Choice of metrics used in collaborative filtering and their impact on recommender systems, in: Proceedings of the IEEE International Conference on Digital Ecosystems and Technologies DEST, 2008, pp. 432–436, 10.1109/DEST.2008.4635147.

[NCF] J.L. Herlocker, J.A. Konstan, J.T. Riedl, An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms, Information Retrieval 5 (2002) 287–310.

[AGG] JAMESON, Anthony. More than the sum of its members: challenges for group recommender systems. In: Proceedings of the working conference on Advanced visual interfaces. 2004. p. 48-54. 334 citations

[PEA] L. Sheugh and S. H. Alizadeh, "A note on pearson correlation coefficient as a metric of similarity in recommender system," 2015 AI & Robotics (IRANOPEN), Qazvin, Iran, 2015, pp. 1-6, doi: 10.1109/RIOS.2015.7270736.