1 Introduzione

L'obiettivo della tesina è sviluppare un algoritmo che confronti due sequenze genetiche qualsiasi. Ogni sequenza è memorizzata come una successione di caratteri nell'insieme $\{A, C, G, T\}$. Per farlo, però, l'algoritmo non confronta brutalmente le due sequenze ma prima ne determina i k-mer, registrando la frequenze con cui compaiono nella sequenza, e poi vengono confrontate con una distanza che quantifica la similarità/dissimilarità tra due sequenze. Un k-mer è l'insieme di tutte le possibili sottostringhe di caratteri consecutivi di lunghezza k; ad esempio, se la nostra sequenza è del tipo "AGAT" e $k = 3$, i k-mer saranno "AGA" e "GAT". Le distanze prese in considerazione sono la distanza euclidea e la distanza D2; dati due vettori u e v di dimensione n, la distanza euclidea è definita al solito come

$$\text{euclidean}(u, v)^2 = \sum_{i=1}^{n} (u_i - v_i)^2,$$

mentre la distanza D2 è definita come

$$D2(u, v) = \sum_{i=1}^{n} u_i v_i.$$

L'idea di base è che sequenze simili avranno anche k-mer simili e le operazioni matematiche con le frequenze delle parole danno una buona misura relativa della dissimilarità di sequenza.

L'approccio utilizzato è il map-reduce che consta di due fasi
• Map(K,V): data in input una coppia chiave-valore (K, V), la funzione map restituisce in output un insieme di coppie intermedie (K_i, V_i)
• Reduce($K, \{V_1, V_2, ..., V_n\}$): dato in input un insieme di valori intermedi $V_1, V_2, ..., V_n$ e una chiave K, la funzione reduce aggrega i valori che condividono la stessa chiave.

2 Procedura Map-Reduce

Nel nostro esempio specifico l'algoritmo si suddivide nelle seguenti fasi:
• Fase 0 (Loading):
Il file viene caricato e letto in formato fasta. Esso è formato da una sequenza molto lunga di caratteri alfabetici.

• Fase 1 (Splitting):
Successivamente si definisce una funzione k-mers che, fissato k, determina tutte le sotto-stringhe di lunghezza k di caratteri consecutivi del file in input.

• Fase 2 (Map-Reduce):
A questo punto si passa alla fase nella quale vengono implementate le operazioni di map e reduce: prima viene eliminata la prima riga che contiene unicamente il nome del file, poi con un'operazione map il file viene diviso in righe di testo, segue una fase di pulizia delle righe, e a quel punto determiniamo tutti i possibili k-mer e le relative frequenze. Questa fase viene applicata a due file distinti. In

questo modo otteniamo due RDD distinte

- Fase 3 (Union):

Le due RDD ottenute nella fase precedente vengono unite con un'operazione di union e, se vogliamo implementare la distanza euclidea, si deve determinare la differenza al quadrato tra due frequenze che condividono la stessa chiave, se invece vogliamo implementare la distanza D2, si deve determinare il prodotto tra due frequenze aventi la stessa chiave con un'operazione di reduceByKey. Per finire si implementa una funzione di somma.

La procedura ricorda molto il metodo del word counting con la differenza che vengono determinate le occorrenze delle sotto-stringhe di lunghezza k nella stringa, inoltre successivamente viene applicata una funzione che ne determina la distanza ossia la similarità/dissimilarità tra le due stringhe.

2.1 Scelta della taglia k-mer

La scelta della dimensione k-mer ha molti effetti diversi sull'assemblaggio della sequenza. Questi effetti variano notevolmente tra i k-mer di dimensioni inferiori e di dimensioni maggiori. Pertanto, è necessario ottenere una comprensione delle diverse dimensioni k-mer per scegliere una dimensione adatta che bilancia gli effetti. Come regola pratica, i k-mer più piccoli dovrebbero essere usati quando le sequenze sono ovviamente diverse (ad esempio, non sono correlate) mentre i k-mer più lunghi possono essere usati per sequenze molto simili.

Per le funzioni basate su istogramma, la scelta dei k-mer è cruciale per il successo di tali metodi. È una scelta euristica. È da rimarcare che anche la scelta di k dovrebbe essere in un intervallo che conservi il "contenuto teorico dell'informazione" delle sequenze da analizzare.

3 Algoritmo in Pyspark

```
def k_mers(s,k):
  kmers=[]
  for i in range(len(s)-k+1):
    kmers.append(s[i:i+k])
  return kmers
```

riporta l'algoritmo che determina le sottostringhe consecutive di lunghezza k di una stringa di caratteri. Ad esempio, se applichiamo la funzione alla stringa "GTAGAGCTGT" fissando k = 5, otteniamo

$$\{\text{GTAGA}, \text{TAGAG}, \text{AGAGC}, \text{GAGCT}, \text{AGCTG}, \text{GCTGT}\}$$

Di seguito è riportato il codice applicato ad un esempio.

```
b1 = sc.textFile("NC_009057.fasta")
b2 = sc.textFile("NC_009063.fasta")


kmer1 = b1.filter(lambda x: '>NC_009057' not in x)
kmer1=kmer1.map(lambda x: x.lower().strip(',.;:!?()-')).flatMap(lambda x: k_mers(x,15))
kmer1=kmer1.map(lambda x: (x,1)).reduceByKey(lambda x,y: x+y)

kmer2 = b2.filter(lambda x: '>NC_009063' not in x)
kmer2=kmer2.map(lambda x: x.lower().strip(',.;:!?()-')).flatMap(lambda x: k_mers(x,15))
kmer2=kmer2.map(lambda x: (x,1)).reduceByKey(lambda x,y: x+y)


# Distanza euclidea
kmer=kmer1.union(kmer2).reduceByKey(lambda x,y: (x-y)*(x-y))

print(m.sqrt(kmer.values().sum()))

# Distanza D2
kmer=kmer1.union(kmer2).reduceByKey(lambda x,y: x*y)

print(kmer.values().sum())
```

La prima parte ricorda molto la funzione word counting con la differenza che all'inizio, attraverso la funzione *map*, associamo la frequenza 1 ad ogni sottostringa di lunghezza k di caratteri consecutivi e poi, con la funzione *reduceByKey*, vengono sommate le occorrenze che condividono la stessa chiave ossia, nell'esempio, lo stesso k-mer.

3.1 Risultati Pyspark

Di seguito sono riportate le tabelle che che conservano i risultati dell'analisi. Le tabelle sono tabelle a doppia entrata che riportano i nomi delle stringhe. k è stato fissato a 15. La prima tabella riporta i valori ottenuti applicando la distanza euclidea alle stringhe

Euclidea	NC-009057	NC-009058	NC-009059	NC-009060	NC-009062	NC-009063
NC-009057	0	155.31	156.30	157.56	144.72	142.49
NC-009058	155.31	0	157.22	157.98	156.88	156.48
NC-009059	156.30	157.22	0	151.50	156.78	156.76
NC-009060	157.56	157.98	151.50	0	158.04	157.9
NC-009062	144.72	156.88	156.78	158.04	0	140.39
NC-009063	142.49	156.48	156.76	157.9	140.39	0

Notiamo che nell'esempio le due sequenze più simili, ossia quelle di distanza euclidea minima tra le sequenze considerate, sono "NC-009062" e "NC-009063".

La seconda tabella riporta i risultati ottenuti applicando la distanza D2 alle stringhe

D2	NC-009057	NC-009058	NC-009059	NC-009060	NC-009062	NC-009063
NC-009057	0	25333	25470	25686	23735	23435
NC-009058	25333	0	25593	25731	25666	25504
NC-009059	25470	25593	0	24710	25533	25531
NC-009060	25686	25731	24710	0	25750	25729
NC-009062	23753	25566	25533	25750	0	23125
NC-009063	23435	25504	25531	25729	23125	0

Notiamo che, scegliendo la distanza D2 come misura di similarità tra due sequenze genetiche, le due sequenze più simili sono sempre "NC-009062" e "NC-009063".

4 Algoritmo in Dask

Anzitutto viene implementata una funzione che determina il vettore dei k-mer.

```
def k_mers(s,k):
  kmers=[]
  for i in range(len(s)-k+1):
    kmers.append(s[i:i+k])
  return kmers
```

La figura

```
b1 = db.read_text("NC_009057.fasta")
b2 = db.read_text("NC_009063.fasta")

kmer1=b1.filter(lambda x: "NC_009057" not in x).str.replace(".","").str.lower().str.strip(
kmer1=kmer1.map(lambda x: k_mers(x,15)).flatten().frequencies(sort=True)

kmer2=b2.filter(lambda x: "NC_009063" not in x).str.replace(".","").str.lower().str.strip(
kmer2=kmer2.map(lambda x: k_mers(x,15)).flatten().frequencies(sort=True)
```

mostra il codice dell'algoritmo applicato a due stringhe. Anzitutto la prima riga del testo viene eliminata, poi viene rimossa la punteggiatura, gli "/n" e gli spazi finali. Successivamente viene applicata al testo la funzione k-mers definita come nella prima parte relativa a Pyspark e per finire vengono concatenati tutti gli elenchi di parole e vengono calcolate le occorrenze. A quel punto possiamo stampare le sotto-stringhe con le relative frequenze nella stringa.
Questa funzione ricorda molto la funzione word counting con l'unica differenza che, al posto di determinare le frequenze delle parole, vengono determinate le occorrenze dei k-mer.

```
def sub(x, y):
  if x[0]==y[0]:
     return (x[1] - y[1])*(x[1]-y[1])

kmer=db.concat([kmer1,kmer2])
kmer.compute()

s=kmer.foldby(lambda x: x[0],sub).filter(lambda x: x!=None)
s=s.map(lambda x: x[1]).sum().compute()
m.sqrt(s)
```

mostra l'algoritmo che determina la distanza euclidea tra due stringhe. Anzitutto si definisce una funzione che determina la differenza al quadrato tra due valori se questi posseggono la stessa chiave, poi le due bag vengono unite con la funzione *concat* che è l'equivalente di union per Dask, infine attraverso *foldby*, che è l'equivalente di *reduceByKey* per Dask, si calcola la differenza dei valori per tutte le coppie che condividono la stessa chiave per poi sommarle. Quello che si ottiene è la distanza euclidea al quadrato.

Analogamente è riportato l'algoritmo che determina la distanza D2

```
def molt(x, y):
  if x[0]==y[0]:
   return x[1]*y[1]

kmer=db.concat([kmer1,kmer2])
kmer.compute()

s=kmer.foldby(lambda x: x[0],molt).filter(lambda x: x!=None)
s=s.map(lambda x: x[1]).sum().compute()
s
```

5 Confronto tra PySpark e Dask

In Dask l'implementazione dell'algoritmo è più laboriosa dal momento che rispetto a Pyspark ha meno funzioni già preinstallate e anche le prestazioni risultano migliori in Pyspark.

Riferimenti bibliografici

[1] Umberto Ferraro Petrillo ,Francesco Palini , Giuseppe Cattaneo , Raffaele Giancarlo, *Alignment-free Genomic Analysis via a Big Data Spark Platform.*

[2] Andrzej Zielezinski, Susana Vinga, Jonas Almeida, Wojciech M. Karlowski, *Alignment-free sequence comparison: benefits, applications, and tools.*

[3] *https://en.m.wikipedia.org/wiki/K-mer.*