Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

Studio della filogenesi di molecole di RNA attraverso l'uso della rappresentazione k-mer counting

Laureando
Francesco Allevi

Matricola 000000

Relatore
Relatore Name

Correlatore
Correlatore Name

A.A. 2018/2019

Indice

1. Introduzione

Questo progetto tratta della realizzazione di un framework in python per il conteggio dei k-mer e la selezione dei k-mer interessanti per il riconoscimento della filogenesi.

1.1 Motivazione

1.2 Scopo del Progetto

Lo scopo del progetto è di costruire un framework in **python** per il riconoscimento della filogenesi di una sequenza di RNA. Il progetto, in particolare, cerca di predire il **Phylum**, la **Classe**, e l'**ordine** di una sequenza. Il framework è composto da due parti: la prima parte del framework servirà a calcolare i k-mer di una sequenza genetica. La seconda parte del framework serve per ridurre i k-mew che non sono utili al riconoscimento della filogenesi. Il riconoscimento della filogenesi avverrà tramite i k-mer presenti all'interno della stessa.

1.3 Contesto Generale

In questa sezione, in primo luogo, verrà spiegato quale alfabeto viene preso in considerazione per le sequenze di RNA.Successivamente verrà spiegato che cosa è un k-mer e come si può estrarre un k-mer da una sequenza di RNA.

Alfabeto Accettato

In una sequenza di RNA vengono accettati i seguenti caratteri.

Carattere	**Significato**
"A"	Adenosina
"C"	Citosina
"G"	Guanina
"U"	Uracile
"R"	Adenosina o Guanina
"Y"	Citosina o Uracile
"S"	Guanina o Citosina
"W"	Adenosina o Uracile
"K"	Guanina o Uracile
"M"	Adenosina o Citosina
"B"	Citosina o Guanina o Uracile
"D"	Adenosina o Guanina o Uracile
"H"	Adenosina o Citosina o Uracile
"V"	Adenosina o Citosina o Guanina
"N"	Qualsiasi base
"." o "-"	gap

Definizione ed esempi di k-mer e della loro frequenza

Data una sequenza biologica, un k-mer è una sottostringa della medesima, di lunghezza k . Quando k è di lunghezza uguale ad 1, i k-mer vengono chiamati monomeri. Il conteggio della frequenza dei k-mer viene utilizzato nella bioinformatica per analizzare le sequenze di DNA/RNA. Nel nostro caso vengono analizzati i k-mer delle sequenze di RNA per ricostruire la filogenesi della sequenza tramite l'utilizza di un modello di machine learning. Per filogenesi si intende il processo di ramificazione delle linee di discendenza. Il conteggio dei k-mer può essere utilizzato come firma della sequenza analizzata. Infatti il numero di k-mer presenti in una sequenza di lunghezza **L** è uguale a

$$L - k + 1$$

Esempio dei k-mer all'interno di una sequenza di RNA:
Prendiamo come esempio la sequenza:"ACGACUYN". Mentre come valore k prendiamo uno, inizialmente. La sequenza è di lunghezza *8*, di conseguenza il numero dei k-mer presenti nella sequenza è di 8. Il primo k-mer individuato è il k-mer "A" (il primo carattere all'interno della sequenza). Il k-mer "A" avrà frequenza due, visto che è presente due volte all'interno della sequenza. Il prossimo k-mer di lunghezza uno presente è "C", con frequenza due. Ecco l'elenco di tutti i k-mer con la loro frequenza di lunghezza 1:

- A:2
- C:2
- G:1
- U:1
- Y:1
- N:1

Successivamente si imposterà il valore k a due. Di conseguenza il numero di k-mer con k uguale a due è di sette. Per k uguale a due il primo k-mer disponibile è "AC", con frequenza due. Il secondo k-mer della sequenza sarà "CG" che ha frequenza uno. Di seguito l'elenco con la frequenza di tutti i k-mer di lunghezza due della sequenza "ACGACUYN":

- AC:2
- CG:1
- GA:1
- CU:1
- UY:1
- YN:1

E così via fino al valore k desiderato.

2. Algoritmi Per Kmer

In questo capitolo si andrà a discutere degli algoritmi utilizzati per il calcolo dei k-mer. Gli algoritmi che abbiamo implementato sono presenti nell'articolo scientifico presente nella bibliografia [4]. Secondo lo studio, i migliori algoritmi per il calcolo dei k-mer sono **DSK**, **Gerbil** e **KMC3**. Tutti e tre gli algoritmi sono detti "disk based", ovvero fanno affidamento sul disco fisso per cercare un buon compromesso tra l'utilizzo della memoria principale utilizzata e tempo di esecuzione. Di seguito verrano spiegati il loro funzionamento e la loro implementazione.

2.1 DSK

L'algoritmo DSK(disk streaming k-mer) [1] è un algoritmo per il calcolo dei k-mer in cui viene decisa dall'utente la quantità di memoria e spazio sul disco questo approccio attua un compromesso tra memoria, disco e tempo. Il multiset di tutti i k-mer viene partizionato e ogni partizione viene salvata nel disco poi, ogni partizione viene caricata separatamente in memoria in una hash table temporanea. Il conto dei k-mer esce fuori ripercorrendo ogni hash table. L'algoritmo prevede una prima fase in cui vengono calcolati:

- il numero di k-mer presenti in una sequenza.
- il numero d'iterazioni necessarie a leggere, salvare su disco e contare tutti i k-mer e la loro frequenza nella sequenza.
- Il numero di partizioni necessarie per ogni iterazione.

Ad ogni iterazione vengono letti tutti i k-mer e **se è l'iterazione giusta**, il k-mer letto viene salvato in una delle partizioni create. La formula utilizzata per decidere se salvare o no il k-mer in una partizione è se il k-mer m rispetta il seguente predicato:

$$(h(m) \mod N_{iters}) == i \tag{2.1}$$

dove:

- m è il kmer letto.
- h(m) è una funzione di hash applicata al k-mer m.
- Niters è il numero totale d'iterazioni.
- i è l'iterazione attuale.

Se il k-mer rispetta il predicato soprastante (2.1) allora verrà salvato nella partizione j. La formula per determinare in quale partizione salvare il k-mer è la seguente:

$$j = (h(m)/N_{iters}) \mod N_p \tag{2.2}$$

dove:

- j è la partizione attuale.
- m è il k-mer attuale.
- Niters è il numero d'iterazioni totali.
- h(m) è la funzione di hash applicata al k-mer m.
- Np è il numero totale di partizioni necessarie.

Calcolo del numero di iterazioni

Il numero d'iterazioni è dato dalla sequente formula:

$$N_{iters} = \lceil \frac{v * 2^{\log_2 2k}}{D} \rceil$$

Dove:

- v è il numero di k-mer in una sequenza.
- k è la dimensione dei k-mer.
- D è dimensione su disco occupata dalle partizioni.

Calcolo del numero di Partizioni necessarie ad ogni Iterazione:

Il numero di partizioni necessarie ad ogni iterazione è dato da:

$$N_p = \lceil \frac{v * (2^{\lceil \log_2 2k \rceil} + 32)}{0.7 * N_{iters} * M} \rceil$$

Dove:

- v è il numero di k-mer presenti nella sequenza.
- k è la dimensione dei k-mer.
- N_iters è il numero d'iterazioni necessarie.
- M è la dimensione in memoria che l'algoritmo utilizzerà ad ogni iterazione.

Si noti che a ogni iterazione non è detto che vengano utilizzate tutte le partizioni per salvare i k-mer.

2.1.1 Descrizione del codice utilizzato per l'implementazione:

Di seguito verrà mostrata l'implementazione della lettura e del salvataggio nelle partizioni dell'algoritmo **DSK**.

```
while kmer_reader.has_next(k_number):
  kmer = kmer_reader.read_next_kmer()
  dsk_utils = DefaultDSKUtils(j, kmer)
  dsk_utils.set_partition_number(partition_number)
  dsk_utils.set_iteration_number(iteration_number)
  if dsk_utils.equals_to_ith_iteration():
    dsk_utils.set_partition_index()
    path = os.path.join(self.__partition_path,filename.split(".")[0])
    path = os.path.join(path, "partition-"+
                  str(dsk_utils.get_partition_index())+".bin")
    dsk_utils.write_to_partitions(path,kmer)
```

La classe "DefaultDSKUtils" implementa le formule (2.1) e (2.2) e si occupa di salvare nelle partizioni il k-mer. Infatti a ogni k-mer letto, l'oggetto creato dalla classe "DefaultDSKUtils" gli viene passato il kmer letto e il numero dell'iterazione attuale. Poi all'oggetto creato viene assegnato il numero totale d'iterazioni e di partizioni create. Il predicato (2.1) viene implementato dal metodo "equals_to_ith_iteration", mentre la formula (2.2) è implementata dal metodo "set_partition_index". La funzione di hash utilizzata è **MD5**. Una volta che i k-mer sono salvati nelle partizioni verranno letti e poi salvati nel file di uscita:

```
for j in range(partition_number):
  hash_table = self.initialize_dict()
  path = os.path.join(self.__partition_path, filename)
  path = os.path.join(path, "partition-" + str(j) + ".bin")
  partition_kmer_reader = PartitionKmerReader(path, self.__k)
  size = partition_kmer_reader.get_file_lenght()
  while partition_kmer_reader.has_next(size):
    m = partition_kmer_reader.read_next_kmer()
    s = m.decode("utf-8")
    if s in hash_table:
      hash_table[s] = hash_table[s] + 1
    else:
      hash_table[s] = 1
  if os.path.exists(path):
    os.remove(path)
    out_writer = OutputWriter(filename=molecule_name
                              , path=self.__out_path)
    hash_table = self.__sort_dictionary(hash_table)
    out_writer.write_to_output(hash_table)
    out_writer.close_all_files()
```

Nel codice soprastante si leggono man mano i k-mer e si inseriscono temporaneamente in un dizionario la cui chiave corrisponde a un k-mer e il valore la sua frequenza.

2.2 Gerbil

Gerbil [3] , è un algoritmo che legge i k-mer, li salva nelle partizioni, poi li conta e salva nel file di uscita. Gerbil per risparmiare lo spazio occupato dalle partizioni utilizza i minimizer. Gerbil è un algoritmo che lavora in modo più efficiente quando la lunghezza dei k-mer è maggiore o uguale di 32. È un algoritmo che è diviso in più fasi:

1. viene letto un k-mer.Dal k-mer letto viene estratto il super k-mer.il super k-mer viene salvato in una partizione.
2. Vengono letti i super k-mer dalle partizioni, vengono trasformati in k-mer, vengono contati e salvati nel file di output.

2.2.1 Minimizers e Super k-mer

I minimizer sono un modo per salvare più k-mer contigui risparmiando spazio. Un super k-mer è una sottostringa di un k-mer. È infatti grazie al super k-mer che i minimizer funzionano.

Esempio:

Si consideri la sequenza "ACGACCUNNACCC" e il valore k uguale a 4. La sequenza è composta dai seguenti k-mer: ACGA,CGAC,GACC,ACCU,CCUN,CUNN,UNNA, NNAC,NACC,ACCC. Nei k-mer contigui ci sono alcune parti ripetute. Ad esempio i k-mer ACGA e CGAC hanno la sottostringa "CGA" in comune. Per estrarre i super k-mer da un k-mer bisogna prima decidere prima la lunghezza del minimizer. Il minimizer determina quale quanti nucleotidi dei k-mer contigui verranno eliminati. I super k-mer di lunghezza 2 estratti dalla sequenza con k uguale a quattro sono:

minimizer	**super k-mer**
GA	AA,CC,CC
CU	AC,CN,NN
NA	UN,NC,CC
CC	AC

Se k è la lunghezza del k-mer e m è la lunghezza del minimizer, allora il numero massimo di super k-mer contigui che quel minimizer può memorizzare è uguale a:

$$k - l + 1 \tag{2.3}$$

2.2.2 Implementazione di Gerbil:

Di seguito l'implementazione della prima parte di Gerbil:

```
minimizer = ""
gerbil_utils = DefaultMinimizerHandler(self.__k, self.__m)
kmer_list = list()
while reader.has_next(size):
    kmer = reader.read_next_kmer()
    if min_ith == 0:
        minimizer = gerbil_utils.get_minimizers_from_kmer(kmer)
        min_ith += 1
```

```
            kmer_list.append(kmer)
        elif min_ith == self.__super_kmer_length - 1:
            kmer_list.append(kmer)
            gerbil_utils.find_super_kmer_and_write(kmer_list
                                                   , minimizer,
                                                partition_file_path)
            min_ith = 0
            kmer_list.clear()
        else:
            min_ith += 1
            kmer_list.append(kmer)
    if len(kmer_list) > 0:
        gerbil_utils.find_super_kmer_and_write(kmer_list
                                               , minimizer
                                               , partition_file_path)
```

La variabile "minimizer" è una stringa inizialmente vuota, ma che servirà come appoggio temporaneo ai minimizer trovati. L'oggetto "gerbil_utils" serve a gestire i vari minimizer e super k-mer. Con il metodo "get_minimizers_from_kmer" è possibile estrarre un minimizer da un k-mer. Esso restiuisce gli ultimi m caratteri da un k-mer. Successivamente quando si saranno letti il numero dei k-mer ricavati dalla formula (2.3), questi verranno passati insieme al minimizer al metodo "find_super_kmer_and_write", che si occuperà di eliminare dai k-mer letti il minimizer e di creare (se non esiste già) e salvare nella partizione i super k-mer generati. Se ad esempio la sequenza è uguale ad "AACACG". Se il minimizer è "CA" (visto che il primo k-mer letto è "AACA"), allora in "find_super_kmer_and_write" verranno creati i super k-mer AA,AC,CG. I super k-mer saranno scritti nelle partizioni tutti attaccati (quindi nel nostro esempio diventano "AAACCG"), mentre il minimizer verrà usato nel nome della partizione. Se la partizione esiste già vuol dire che precedentemente è già stato trovato un altro minimizer identico. Di seguito l'implementazione della seconda parte di Gerbil:

```
while reader.has_next(size - 1):
    kmer = reader.read_next_kmer()
    if kmer in hash_table:
        d[kmer] = d[kmer] + 1
    else:
        d[kmer] = 1
```

Questo pezzo di codice si occupa di leggere i super k-mer dalle partizioni, rimettere i minimizer al loro posto e salvarlo in un dizionario, dove la chiave è il k-mer, mentre il valore la frequenza. La lettura e il ricomponimento del k-mer è effettuato dall'oggetto "reader", che appartiene alla classe "SuperKmerReader". L'oggetto conosce il minimizer visto che è il nome della partizione e conosce anche quanto è lungo il super k-mer. Lui sa che al primo super k-mer letto, il minimizer deve essere aggiunto alla fine. Il secondo super k-mer, il minimizer deve essere aggiunto tra il penultimo e l'ultimo carattere. Il terzo super k-mer, il minimizer deve essere aggiunto tra il terzultimo e il penultimo carattere. Di seguito implementation del metodo "read_next_kmer" della classe "SuperKmerReader":

```
super_kmer = self.__file.read(self.__super_kmer_size)
        super_kmer = super_kmer.decode("utf-8")
        if self.__ith == self.__super_kmer_size:
```

```
            kmer = super_kmer + self.__minimizer
        elif self.__ith == 0:
            kmer = self.__minimizer + super_kmer
        else:
            kmer = super_kmer[:self.__ith] + self.__minimizer
                                    + super_kmer[self.__ith:]
        self.__ith -= 1
        if self.__ith < 0:
            self.__ith = self.__super_kmer_size
        self.__size_counter += 1
        return kmer
```

La variabile "self.__ith" serve per sapere in quale posizione inserire il minimizer quando si legge il super k-mer. Inizialmente sarà impostata all'ultimo carattere, poi man mano che i super k-mer verranno letti, questa scala fino al primo carattere. In questo caso se nella partizione ci sono ancora super k-mer la variabile viene impostata di nuovo all'ultimo carattere.

2.3 KMC3

KMC3 [2] è un algoritmo per il conteggio della frequenza dei k-mer. Questo algoritmo è molto simile a gerbil. La più grande differenza è che le partizioni, nella seconda fase, vengono letti in base all'ordine lessicografico dei minimizer. L'algoritmo utilizza, per ordinare i minimizers, l'algoritmo **"Most Significant Radix Sort"** (da qui in avanti verrà chiamato MS radix Sort). KMC3 è diviso in due fasi:

1. viene letto un k-mer.Dal k-mer letto viene estratto il super k-mer.il super k-mer viene salvato in una partizione.

2. Vengono letti i minimizer delle partizioni, vengono ordinati e vengono letti i super k-mer dalle partizioni nell'ordine trovato.Poi vengono trasformati in k-mer, vengono contati e salvati nel file di output.

2.3.1 implementazione della prima fase:

In questo caso il codice è identico alla prima fase di gerbil.

```
minimizer = ""
    kmer_list = list()
    mh = DefaultMinimizerHandler(self.__k, self.__m)
    while reader.has_next(length):
        kmer = reader.read_next_kmer()
        if min_ith == 0:
            kmer_list.append(kmer)
            minimizer = mh.get_minimizers_from_kmer(kmer)
            min_ith += 1
        elif min_ith == self.__m - 1:
            kmer_list.append(kmer)
            mh.find_super_kmer_and_write(kmer_list, minimizer, partition_su
            kmer_list.clear()
            min_ith = 0
```

```
            else:
                kmer_list.append(kmer)
                min_ith += 1
        if len(kmer_list) > 0:
            mh.find_super_kmer_and_write(kmer_list, minimizer, partition_sub
```

Il modo in cui i super k-mer e i minimizers vengono letti e creati è lo stesso di Gerbil.

2.3.2 implementazione della seconda fase:

In questa fase le partizioni vengono lette nell'ordine dettato da "MS radix Sort". MS radix sort, come si può dedurre dal nome, inizia ad ordinare le stringhe a partire dal carattere più significativo. L'algoritmo di ordinamento è stato implementato nel seguente modo dalla classe "MostSignificantRadixSort":

```
exp = self.__string_length-1
    i = 0
    while exp > i:
        self.counting_sort(i)
        i += 1
```

Come si può notare l'algoritmo utizza un altro algoritmo di ordinamento per ordinare le stringhe in base al carattere che si sta analizzando, ovvero **"Counting Sort"**. Una volta che si ha l'ordine in cui le partizioni devono essere lette, i super k-mer vengono trasformati in kmer dalla classe "SuperKmerReader" (descritta nel capitolo 2.2.2.) Successivamente vengono inseriti in un dizionario in cui la chiave è il k-mer, mentre il valore corrisponde alla frequenza di quel k-mer. Di seguito l'implementazione della seconda fase:

```
size = reader.get_file_lenght()
        ht = dict()
        while reader.has_next(size - 1):
            kmer = reader.read_next_kmer()
            if kmer in ht:
                ht[kmer] += 1
            else:
                ht[kmer] = 1
            if len(ht) > 64:
                self.__write_kmer_count(part_name, ht)
                ht.clear()
        if len(ht) > 0:
            self.__write_kmer_count(part_name, ht)
            ht.clear()
```

Come si può notare al massimo verranno letti 64 k-mer alla volta. Questo è stato fatto per non occupare troppa memoria principale. Infatti una partizione può contenere molti super k-mer.

2.4 Formato del file di uscita

Il file di output è in formato "CSV". Nell'intestazione sono presenti i k-mer trovati in almeno una le molecole analizzate. Se ad esempio la molecola A ha il k-mer "CC" e

la molecola B non lo ha, questo verrà comunque inserito nell'header. Il primo valore nell'intestazione è l'id. L'id corrisponde al nome della molecola. Dopo l'id, nell'intestazione sono presenti i k-mer messi prima in ordine di lunghezza e poi in ordine alfabetico. Ad esempio il k-mer "U" verrà inserito prima del k-mer "AA".

2.4.1 Esempio del file di uscita:

Prendiamo per esempio la molecola di nome "A" e la molecola di nome "B". La molecola A possiede i seguenti k-mer:

- C con frequenza 2
- CA con frequenza 3
- UU con frequenza 1

La molecola B possiede i seguenti k-mer:

- U con frequenza 10
- GC con frequenza 2
- UU con frequenza 3

Allora il file di uscita sarà fatto nel seguente modo:
id,C,U,CA,CG,UU
A,2,0,3,0,1
B,0,10,0,2,3

3. Selezione Delle Features

In questo capitolo andremo a discutere della selezione delle features. Nel nostro caso le features sono i k-mer. L'obiettivo della riduzione delle features è la diminuzione delle features inutili per determinare la filogenesi delle molecole. Per selezionare le features, nella nostra implementazione abbiamo utilizzato la libreria **Scikit** .

3.1 Bassa varianza

Con questo metodo è possibile ridurre le features la cui varianza è minore dell'80%.

3.1.1 implementazione della riduzione delle features a bassa varianza:

L'implementazione è la seguente:

```
def apply_low_variance(self, threshold):
    sel = VarianceThreshold(threshold=threshold)
    arr = sel.fit_transform(X=self.__df)
    self.__selected_features = sel.get_support()
    self.__output_arr = arr
```

Il parametro "threshold" corrisponde al valore minimo della varianza per cui una feature è selezionata.

3.2 LinearSVC con Penalità L1

Questo metodo [**l1-based**] utilizza un modello lineare chiamato "LinearSVC" con una penalità al modello chiamata L1. LinearSVC sta per "Linear Support Vector Classification". Un modello "LinearSVC" è un metodo simile a SVC, ma implementato con una libreria diversa, che permette più flessibilità nella scelta delle penalità e della funzione di loss. Nel nostro caso è stato implementato nel seguente modo:

```
y = np.asarray(list(self.__molecule_expected.values()))
    lsvc = LinearSVC(C=0.1, penalty="l1", dual=False).fit(X=self.__df, y
    model = SelectFromModel(lsvc, prefit=True)
    self.__output_arr = model.transform(X=self.__df)
```

3.3 Alberi di ricorsione

Questo metodo permette di rimuovere le features inutili tramite un classificatore chiamato "ExtraTreeClassificator". Il classificatore si adatta a un numero di alberi decisio-

nali randomizzati su vari sotto campioni del dataset e utilizza la media per migliorare l'accuratezza predittiva e il controllo dell'overfitting.

```
y = np.asarray(list(self.__molecule_expected.values()))
    clf = ExtraTreesClassifier(n_estimators=5)
    clf = clf.fit(self.__df, y)
    model = SelectFromModel(clf, prefit=True)
    self.__output_arr = model.transform(self.__df)
```

3.4 Test del chi2

Il test del chi quadrato è un test statistico. Esso trasforma i valori in valori positivi facendone il quadrato. Chi quadrato misura la dipendenza tra variabili stocastiche, quindi questo metodo elimina le features che hanno maggiori probabilità di essere indipendenti dalla classe.

```
y = np.asarray(list(self.__molecule_expected.values()))
chi2_feat = SelectPercentile(chi2)
self.__output_arr = chi2_feat.fit_transform(self.__df, y)
self.__selected_features = chi2_feat.get_support()
```

Bibliografia

[1] Rayan Chikhi Guillaume Rizk Dominique Lavenier. ≪DSK: k-mer counting with very low memory usage≫. In: *Bioinformatics* (2013). URL: https://academic.oup.com/bioinformatics/article/29/5/652/253092.

[2] Maciej Długosz Marek Kokot Maciej Długosz. ≪KMC 3: counting and manipulating k-mer statistics≫. In: *Bioinformatics* (2017). URL: https://academic.oup.com/bioinformatics/article/33/17/2759/3796399?login=false.

[3] Matthias Müller-Hannemann Marius Erbert Steffen Rechner. ≪Gerbil: a fast and memory-efficient k-mer counter with GPU-support≫. In: *Biomedcentral* (2017). URL: https://almob.biomedcentral.com/articles/10.1186/s13015-017-0097-9.

[4] Shailesh R Sathe Swati C Manekar. ≪A benchmark study of k-mer counting methods for high-throughput sequencing≫. In: *GigaScience* (2018). URL: https://doi.org/10.1093/gigascience/giy125.