INFORMÁTICA APLICADA A LA BIOQUÍMICA EN PYTHON

BIOINFORMATICS IN BIOCHEMISTRY USING PYTHON

Tema: Bioinformática en Python

Trabajo de Fin de Grado en Bioquímica

Autor: **Andrea Morales Montes**

Tutor: **Francisco R. Villatoro Machuca**

Departamento: **Lenguajes y Ciencias de la Computación**

Área de conocimiento: **Ciencia de la Computación e Inteligencia Artificial**

Fecha de presentación: **julio de 2020**

Número de páginas del manuscrito: **30**

Dña. Andrea Morales Montes, con DNI 74743222-F, estudiante del Grado en Bioquímica de la Facultad de Ciencias de la Universidad de Málaga,

DECLARO:

Que he realizado el Trabajo Fin de Grado titulado "Informática Aplicada a la Bioquímica en Python" y que lo presento para su evaluación. Dicho trabajo es original y todas las fuentes bibliográficas utilizadas para su realización han sido debidamente citadas en el mismo.

Para que así conste, firmo la presente en Málaga, el 6 de julio de 2020.

Fdo.: Andrea Morales Montes

Informática Aplicada a la Bioquímica en Python

Andrea Morales Montes

Julio de 2020

Índice general

Índice de figuras

Índice de códigos

Resumen

La bioinformática y el lenguaje Python son dos herramientas informáticas ampliamente usadas entre la comunidad científica. Esto se debe al aumento de datos biológicos disponibles y a la necesidad de métodos capaces de analizarlos. La propuesta de este trabajo es estudiar la viabilidad de la introducción del lenguaje Python en alumnos que están aprendiendo a programar. Para ello, se han realizado las prácticas propuestas al alumnado que cursa la asignatura *Informática Aplicada a la Bioquímica* del Grado en Bioquímica. La sintaxis de los códigos escritos en Python se asemejan mucho a la de R. Sin embargo, la mayor dificultad que se ha encontrado ha sido la diferencia entre librerías de variables de Python y R. Esto hace que se tenga que recurrir a paquetes externos (*pandas* y *NumPy*) que dificultan la realización de los ejercicios propuestos. Se ha visto que la programación en lenguaje Python podría usarse como material complementario y de ampliación para alumnos que quieran profundizar más en la asignatura.

Palabras clave: Bioinformática, Python, R, Biopython, análisis de secuencias

Abstract

The usage of Python computer language in bioinformatics is growing through the scientific community. This is due to the rise of biological available data and the need for scientific methods that can analyse them. This project proposes to study the viability of introducing Python as a computer language in beginner Biochemistry students. Specifically, the current exercises in the subject *Informática Aplicada a la Bioquímica* have been developed. The syntax of the codes in Python was similar to R, currently used in this subject. However, the main problem found was the difference between variable management in Python and in R. As a result, it is necessary to resort to external packages (*pandas* and *NumPy*). This makes more difficult the fulfilment of the exercises. Taking into account my experinece, a possible application could be the usage of my Python codes as extra material for the students that would like to delve into the subject.

Key words: Bioinformatics, Python, R, Biopython, data analysis

Capítulo 1

Introducción

La bioinformática es un área de la ciencia de carácter interdisciplinar. Su principal objetivo es el desarrollo de herramientas tanto analíticas como metodológicas para interpretar y explorar grandes volúmenes de información biológica. Dentro de este campo encontramos el análisis de secuencias y datos genómicos, predicción gen/proteína, construcción de redes biológicas, etc. (Abdurakhmonov, 2016).

La necesidad de este tipo de herramientas se encuentra estrechamente relacionada con la necesidad de softwares capaces de analizar grandes cantidades de información procedentes del DNA, RNA, proteínas y metabolitos. (Package et al., 2000).

1.1. Historia de la bioinformática

En la década de 1960 se registraron los primeros proyectos de investigación donde se usó la bioinformática. Esto se debe a que, en esos años, se comenzó a desarrollar métodos de secuenciación de proteínas de una gran variedad de organismos y para poder estudiarlos se requería de métodos computacionales. Este fue el comienzo del diseño de bases de datos como GenBank en 1974 por George Bell (GenBank, 2019) o "The European Molecular Laboratory" (EMBL) en 1980 (Madeira et al., 2019). Uno de los mayores avances bioinformáticos fue la incorporación de búsquedas basadas en los algoritmos web. Esto permitió a los investigadores la búsqueda y comparación de su DNA de estudio con otras secuencias. Es aquí donde nació la base de datos NCBI (National Center for Biotechnology Information (NCBI), 1988) (Abdurakhmonov, 2016).

En la actualidad, encontramos una gran variedad de herramientas bioinformáticas disponibles para todos los usuarios. Según el Instituto Europeo de Bioinformática (EBI), hay 500 aplicaciones registradas y esta cifra sigue creciendo. El poder almacenar, ordenar, analizar y distribuir la información procedente de la secuenciación ha hecho que la bioinformática se convierta en una herramienta fundamental en la biotecnología moderna (Package et al., 2000).

1.2. Python en la bioinformática

Python es un lenguaje de programación extensamente usado. Se caracteriza por ser fácil de aprender, fácil de leer e interpretar y es viable en múltiples plataformas (Bassi, 2007). Además, cuenta con una gran librería científica, donde se encuentra Biopython (Cock et al., 2009) entre otros. Estas características son importantes porque ayudan al usuario a concentrarse en la esencia del código y no es su compleja sintaxis. La introducción del lenguaje Python en el ámbito científico ha sido reciente, pero su expansión fue rápida debido a su combinación versátil tanto de funciones intrínsecas del lenguaje como de características extrínsecas (Ekmekci et al., 2016).

El poder de Python se amplía gracias al catálogo de paquetes del que dispone. Esto hace que este lenguaje de programación sea mucho más extenso con muchas facilidades que permiten el buen desarrollo del código. Dentro de estos paquetes, los más usados en la comunidad científica son *NumPy* (Van Der Walt et al., 2011), *matplotlib* (Hunter, 2007) y *pandas* (McKinney, 2010) (Perkel, 2015).

Además, Python cuenta con múltiples entornos de desarrollo (*Integrated Development Enviroment, IDE*). No hay un IDE perfecto, sino que este se va a encontrar definido por el trabajo que se va a realizar. Entre los entornos de desarrollo más usados en el ámbito científico encontramos PyCharm (PyCharm, 2019), Spyder (Spyder, 2019) y Komodo (Komodo, 2019). Todos ellos se caracterizan por ser gratuitos y ser compatibles en Windows, Linux y macOS. Además, tienen una interfaz que facilita el desarrollo de códigos (Integrated Development Environments, IDE: in Python, 2019). Sin embargo, Spyder es el entorno de desarrollo más usado en este campo. Esto se debe a que ha sido diseñado para científicos, ingenieros y analistas de datos. Presenta un entorno de desarrollo avanzado de edición, depuración y creación de perfiles. Su interfaz se divide principalmente en dos partes. Por un lado, tiene una ventana de edición del código donde el propio IDE va señalando los posibles errores que este puede tener. Por otro lado, tiene una consola iPython que permite ejecutar el código que se desee sin salir del entorno de desarrollo. Spyder se encuentra integrada en la plataforma *Anaconda distribution* (Spyder, 2019) (Anaconda, 2019).

Capítulo 2

Desarrollo del trabajo

El objetivo principal de este proyecto, de carácter bibliográfico, es implementar el lenguaje computacional Python en la asignatura "Informática Aplicada a la Bioquímica" que se imparte en el grado en Bioquímica. Para ello, se ha elegido Spyder (Spyder, 2019) como entorno de desarrollo y el paquete Biopython (Cock et al., 2009) que contiene herramientas esenciales para el desarrollo de los códigos. Además, para el análisis y disposición de los datos, se han usado los paquetes *pandas* (McKinney, 2010) y *NumPy* (Van Der Walt et al., 2011), y para la representación gráfica de los resultados, el paquete *matplotlib* (Hunter, 2007).

Los códigos escritos y que se comentan en el apartado 2.2 se encuentran disponibles en la página GitHub (Andrea Morales , 2019). Además, están disponibles los enunciados de los ejercicios propuestos, las secuencias que se han analizado y las representaciones gráficas de los resultados de los apartados que se piden. Las secuencias usadas están en formato *fasta* y se han descargado de la base de datos NCBI (National Center for Biotechnology Information (NCBI), 1988).

2.1. Instalación de programas

Los tutoriales de instalación de los programas y paquetes se encuentran en el Anexo A acompañados con imágenes de referencia. Sin embargo, estos pasos están descritos para el sistema operativo Linux Mint (Linux Mint, 2019), el sistema que se ha usado en la escritura del código. En las páginas web citadas en el Anexo A se encuentran tutoriales de instalación para Windows y MacOs.

2.2. Comparación de los lenguajes R y Python en función de las prácticas

2.2.1. Lectura de archivos fasta

Los datos biológicos que se analizan se encuentran en formato fasta. Para poder extraer esa información, se ha usado el paquete `SeqIO` de Biopython. Los fragmentos de código extraídos en la explicación pertenecen a los capítulos 3 y 4 del libro tutorial de Biopython (Chang et al., 2019).

Primero, es necesario cargar el paquete `SeqIO` que se encuentra en el módulo *Bio* de Biopython. Esta función va a pasar el fichero fasta al lenguaje que entiende Python, es decir, va a parsear el archivo.

Si el archivo se compone de una sola secuencia se usa la función `SeqIO.read()`. Tiene dos argumentos: el nombre del archivo y el formato en el que se encuentra. En estas prácticas todos los archivos que se han usado se encuentran en formato fasta, por lo que el segundo argumento siempre va a ser `fasta`.

Código 2.1: Extracto 1 de (Chang et al., 2019)

```
1 from Bio import SeqIO
2 tupan_dna = SeqIO.read("TupanADN.fasta", "fasta")
```

La variable donde se guarda la información del archivo es del tipo Bio.SeqRecord. Para poder obtener información de la secuencia, tenemos distintos atributos que nos la proporcionan:

Código 2.2: Extracto 2 de (Chang et al., 2019)

```
1 tupan_dna.seq ## .seq - proporciona la secuencia
2 tupan_dna.id ## .id - devuelve el numero de acceso de la secuencia
3 tupan_dna.name ## .name - devuelve el nombre de la secuencia
```

La secuencia que se extrae se comporta como un string y pueden aplicarse funciones que se recogen en el paquete Seq. Estas funciones son `.complement()`, `.reverse_complement()`, `.transcription()`, etc.

Si el fichero fasta se compone de más de una secuencia, se usa la función `SeqIO.parse()`. Esta función tiene los mismos argumentos que `SeqIO.read()`. Sin embargo, la principal diferencia entre ambas es la memoria que ocupa. Es por ello que se recomienda el uso de la función `SeqIO.read()` para archivos de una sola secuencia.

Código 2.3: Extracto 3 de (Chang et al., 2019)

```
1 from Bio import SeqIO
2 lsorchid_dna = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
```

La naturaleza de la variable donde se guarda la secuencia es igual que en el caso anterior. Para seleccionar la secuencia que queremos se usan los corchetes y dentro el número de la secuencia (ej. `[21]`). Cabe destacar que en Python, por defecto, se empieza a contar desde 0 y no desde 1, por lo que si queremos acceder a la primera secuencia de la variable tenemos que escribir `[0]` y no `[1]`.Los atributos que se usan para obtener información de la secuencia y la propia secuencia en sí son los mismos que en caso anterior.

En R, la lectura de archivos es más sencilla. No se hace discriminación en función del número de secuencias que compone el archivo, siempre se usa la misma función `read.fasta()`. También, el acceso a información adicional de la secuencia se hace con la función `attr()`, muy similar a como se hace en Python.

2.2.2. Variables del lenguaje

Como se observa en la tabla 2.1, cada lenguaje tiene sus distintos tipos de variables a pesar de que su naturaleza sea muy parecida. En el caso de Python, no encontramos variables del tipo 'matrices' o 'factores', muy importantes en el análisis y visualización de datos biológicos, es por ello por lo que se requieren paquetes externos que nos los proporcionen. Estos paquetes son *pandas* (McKinney, 2010) y *NumPy* (Van Der Walt et al., 2011).

Tabla 2.1: Tipos de variables disponibles en Python y R. Ejemplos (Lutz, 2013) y (Cotton, 2013)

Python		**R**	
Tipo	**Ejemplo**	**Tipo**	**Ejemplo**
Strings	`'spam'`	Vectores	`c(1,2,3,5,6)`
Listas	`[1,[2,'three']]`	Listas	`list(name='Fred', mynumbers=a, mymatrix=y, age=5.3)`
Diccionarios	`{food : 'spam', 'taste': 3}`	Matrices	`matrix(1:20, nrow=5, ncol=4)`
Tuplas	`(1,'spam',4,'U')`	Factores	`factor(sample (letters[1:5], 30, replace = TRUE))`

Los equivalentes de estas variables son `pd.DataFrame()` y `np.array()`. Ambos son sencillos de definir y cuentan con una variedad de argumentos que permiten caracterizarlas. Sin embargo, para poder trabajar con sus datos, (ej. modificarlos, añadir o eliminar elementos, ordenarlos en orden ascendente o descendente, etc.) es necesario utilizar funciones propias del paquete y el resultado tiene una naturaleza distinta a las variables definidas por defecto en Python. En el caso de R, como este tipo de variables se encuentran definidas por defecto en el lenguaje, las funciones que se utilizan para trabajar con

ellas son las propias funciones de R. Esta diferencia hace el trabajo con matrices y factores en Python sea más complejo porque hay cierta incompatibilidad entre las propias variables de Python (ej. strings y listas) y las variables definidas en *pandas* y *NumPy*. En muchas ocasiones, para facilitar el desarrollo de la práctica, se han utilizado listas de listas en vez de `pd.DataFrame`.

Como ejemplo, se muestra el código 2.4, escrito para el ejercicio 3.1 de la práctica 5. En este ejercicio se le pide al alumno que guarde en una matriz, usando un bucle `for()`, todos los nombres y su composición de bases nitrogenadas de todos los genes que se encuentran en el DNA mitocondrial de *Homo Sapiens neanderthalis* (NC_011137.1). En este código, el DNA se encuentra guardado en la variable `record`.

Código 2.4: Práctica 5, ejercicio 3.1

```
1  mitoneannn = [['nombre del gen','A','C','G','T']]
2
3  for i in range(len(record)):
4      temp = record[i]
5      name = temp.name
6      A = temp.seq.count('A')
7      C = temp.seq.count('C')
8      G = temp.seq.count('G')
9      T = temp.seq.count('T')
10
11     mitoneannn.append([name, A, C, G, T])
12
13 Output:
14 [['nombre del gen', 'A', 'C', 'G', 'T'],
15  ['lcl|NC_011137.1_cds_YP_002124302.2_1', 270, 339, 113, 229],
16  ['lcl|NC_011137.1_cds_YP_002124303.1_2', 326, 348, 99, 269],
17  ['lcl|NC_011137.1_cds_YP_002124304.1_3', 417, 461, 250, 414],
18  ['lcl|NC_011137.1_cds_YP_002124305.1_4', 195, 212, 103, 174],
19  ['lcl|NC_011137.1_cds_YP_002124306.1_5', 80, 65, 13, 49],
20  ['lcl|NC_011137.1_cds_YP_002124307.1_6', 204, 227, 73, 177],
21  ['lcl|NC_011137.1_cds_YP_002124308.2_7', 210, 250, 115, 209],
22  ['lcl|NC_011137.1_cds_YP_002124309.1_8', 103, 103, 36, 104],
23  ['lcl|NC_011137.1_cds_YP_002124310.1_9', 82, 91, 38, 86],
24  ['lcl|NC_011137.1_cds_YP_002124311.1_10', 415, 476, 137, 350],
25  ['lcl|NC_011137.1_cds_YP_002124312.1_11', 550, 623, 194, 445],
26  ['lcl|NC_011137.1_cds_YP_002124313.1_12', 102, 37, 188, 198],
27  ['lcl|NC_011137.1_cds_YP_002124314.2_13', 325, 390, 138, 288]]
```

Como se puede observar en el código 2.4, el resultado obtenido usando listas de listas se asemeja mucho al de una matriz. Además, una de las ventajas es que su definición y acceso a los datos es muy sencilla. Si se hubiesen usado los paquetes *pandas* o *NumPy*, el formato del resultado habría sido muy semejante. Sin embargo, el acceso a estos datos es más complejo. Además, no sólo se requiere de un conocimiento previo del lenguaje de Python en sí, sino también saber manejar estos paquetes, sus variables, sus funciones y sus resultados.

2.2.3. Representación gráfica de datos

En Python es necesario descargarse un paquete externo para poder representar datos gráficamente. En este caso se ha elegido *matplotlib* (Hunter, 2007). En R, esto no es necesario, puesto que tiene funciones ya definidas que permiten realizar gráficos.

Para comparar ambos lenguajes, se va a usar el código del ejercicio 1.4 de la práctica 5 (ver código 2.5). En este ejercicio, se le pide al alumno que represente en un diagrama de sectores la composición de bases nitrogenadas del DNA de la mitocondria humana (NC_012920.1). La variable donde se encuentra guardada esta secuencia es `mitoseq`.

Matplotlib es el paquete más usado en la comunidad científica para la representación de datos. La forma de definir los gráficos es muy intuitiva, de forma que se van definiendo en distintas variables las características que va a tener nuestro gráfico. Además, de forma predeterminada, *matplotlib* calcula y dispone en el gráfico los porcentajes de los elementos que se están representando. El resultado se encuentra en la figura 2.1a.

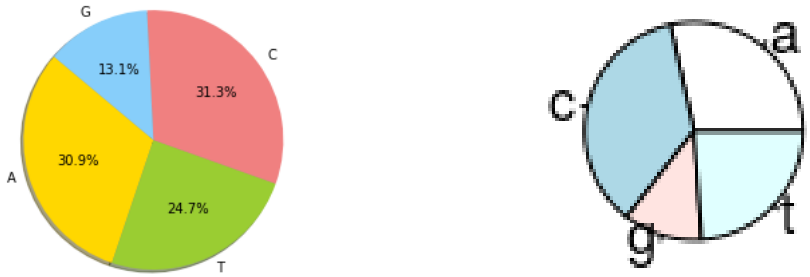
Código 2.5: Práctica 5, ejercicio 1.4

```
1  import matplotlib.pyplot as plt
2
3  l_mitoseq = list(mitoseq)
4
5  count = [l_mitoseq.count('A'),l_mitoseq.count('T'),
6          l_mitoseq.count('C'), l_mitoseq.count('G')]
7
8  labels = ['A', 'T', 'C', 'G']
9  sizes = count
10 colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
11 plt.pie(sizes, labels=labels, colors=colors,
12 autopct='%1.1f%%', shadow=True, startangle=140)
13 plt.axis('equal')
14 plt.savefig('fig1.png',format = 'png')
15
16 plt.show()
```

Por otro lado, el código para obtener este gráfico en R es mucho más sencillo y directo, pero el resultado que se obtiene (figura 2.1b) no es tan visual como la figura 2.1a. Además, para poder representar este diagrama de sectores con sus porcentajes, es necesario calcular los porcentajes aparte y añadirlos posteriormente al gráfico.

Código 2.6: Práctica 5, ejercicio 1.4. (código en R)

```
1  pie(table(mitoseq))
2  png("fig2.png")
```



(a) Resultado en Python. (b) Resultado en R.

Figura 2.1: Representación de la composición de bases nitrogenadas del DNA mitocondrial de Homo Sapiens.

Cabe aclarar que este gráfico (figura 2.1b) puede modificarse y se pueden añadir elementos usando otras funciones adicionales a `pie()`. Este ejemplo es el más sencillo puesto que es por el que se va a comenzar enseñando la representación gráfica al alumnado.

Matplotlib cuenta con un amplio catálogo de gráficos y tutoriales disponibles en su página web (Tutorials - Matplotlib, 2019). En los demás tipos de gráficos usados en estas prácticas, nube de puntos e histogramas, las diferencias son iguales que las comentadas.

2.2.4. Alineamiento de secuencias

Como ejemplo, se ha cogido el ejercicio 1.1 de la práctica 9. En este ejercicio, se le pide al alumno que alinee dos secuencias aplicando el algoritmo de alineamiento global de Needelman-Wunsch y que obtenga la puntuación *score* usando las matrices de puntuación BLOSUM100 y PAM30. El código 2.7 es la implementación en el caso de R:

Código 2.7: Práctica 9, ejercicio 1.1 (código en R)

```
1 pairwiseAlignment("MRTEIESLWVFALASKFNIYMQQHFASLLVAIAITWFTITI",
2 "MEDQVGFGFRPNDEELVGHYLRNKIEGNTSRDVEVAISEVNIC", substitutionMatrix=
3 "BLOSUM100", type="local", gapOpening=-4, gapExtension=-4)
```

Como se observa en el código 2.7, el paquete Biostrings de R tiene una función llamada `pairwiseAligment`. Cuando se ejecuta, muestra en pantalla el alineamiento y la puntuación que este tiene en función de la matriz de substitución que se ha indicado, a través del argumento `substitutionMatrix`. Estas matrices de substitución se encuentran disponibles en una librería interna del paquete Biostrings. Además, cuenta con otros argumentos como son `type`, `gapOpening` y `gapExtension`. El argumento `type` sirve para especificar el tipo de alineamiento que se quiere llevar a cabo (`global`, `local`, `overlap`, `global-local`, y `local-global`). Por otro lado, `gapOpening` y `gapExtension` permiten indicar la penalización que va a tener añadir un espacio o extenderlo en el alineamiento, respectivamente (PairwiseAligment Function, 2019).

En Python, se ha usado la función `Align.PairwiseAligner()` de Biopython (Cock et al., 2009) que se encuentra dentro del paquete `Align` de Bio. Esta función implementa

tanto el algoritmo de alineamiento de Needelman-Wunsch como el de Smith-Waterman, Gotoh y Waterman-Smith-Beyer, de manera global y local (Chang et al., 2019). Por lo que el código sería el siguiente:

Código 2.8: Práctica 9, ejercicio 1.1

```
from Bio import Align
aligner = Align.PairwiseAligner()

s = str('MRTEIESLWVFALASKFNIYMQQHFASLLVAIAITWFTITI')
t = str('MEDQVGFGFRPNDEELVGHYLRNKIEGNTSRDVEVAISEVNIC')

from Bio.SubsMat.MatrixInfo import blosum100
aligner.substitution_matrix = blosum100

score = aligner.score(s, t)
alignments = aligner.align(s, t)
```

Una ventaja de usar la función `Align.PairwiseAligner()` es que tenemos una variable (`aligner`) donde se encuentra almacenada todos los parámetros e información relacionada con el alineamiento de las secuencias. Además, Biopython también cuenta con una librería de matrices de puntuación que se encuentran recogidas en `Bio.SubsMat.MatrixInfo`.

En ambos lenguajes computacionales, el procedimiento es muy similar. Las únicas diferencias a destacar son la posibilidad que proporciona el paquete Biostrings en R de poder indicar el tipo de alineamiento que se quiere realizar y, en Biopython, la variable `aligner` que recoge toda la información que relacionada con el alineamiento.

2.2.5. Cadenas ocultas de Markov

Se ha escogido el ejercicio 1.1 de la práctica 7 como ejemplo para ver las cadenas ocultas de Markov. En este apartado se le pide al alumno que defina una función, llamada `estimaHMM`, que a partir de una secuencia (`sec`) y sus estados ocultos (`hid`) calcule las probabilidades iniciales (`pr`) y las matrices de transición (`T`) y emisión (`E`) de dicha secuencia. El código 2.9 que define la función en R comienza comprobando si `sec` y `hid` tienen la misma longitud, de lo contrario no se puede estimar el modelo de Markov oculto de la secuencia. En el caso de que tengan el mismo tamaño, se comienza calculando las probabilidades iniciales de los distintos estados ocultos. Posteriormente, se calcula la matriz de transición que se va a tener tantas filas y columnas como estados ocultos tenga la secuencia. Y por último, se estudia la matriz de emisión que se va a componer de las probabilidades de cada estado visible asociadas a cada estado oculto. Como resultado, la función `estimaHMM` devuelve una lista compuesta por estos tres elementos: `pr`, `T` y `E`.

Código 2.9: Práctica 8, ejercicio 1.1 (código en R)

```
estimaHMM = function(sec,hid,alph=words(1))
{ require(seqinr)
  if (length(sec)!=length(hid))
    stop("Tam. hid diferente de sec")

  S = alph; nS = length(S);
  H = levels(factor(hid)); nH = length(H);

  pr = count (hid, 1, freq=TRUE, alphabet=H);
  names(pr)=H;

  pT = count(hid,2,freq=TRUE,alphabet=H);
  T = matrix( pT, nH, nH, byrow=TRUE);
  T = T / apply(Tr,1,sum);
  rownames(T)=H; colnames(T)=H;

  E = matrix(,nH,nS); rownames(E)=H; colnames(E)=S;
  E = t(sapply(H, function(h)
        count( sec[hid==h],1,freq=TRUE,alphabet=S)));

  return( list ( p=pr, T=T, E=E ) )}
```

En el caso de Python (código 2.10), el orden y el procedimiento para calcular la cadena de Markov oculta es igual que en el código escrito en R (2.9). Sin embargo, hay una diferencia clara y es que Python al no tener variables del tipo matriz integradas por defecto, se tiene que usar el paquete *pandas*. Se ha intentado calcular sin usar variables del tipo `pd.DataFrame`, pero el resultado obtenido eran variables en las que el acceso a los datos era muy complejo. Además, esto implica que las funciones usadas para moverse dentro de las matrices, acceder a los datos recogidos y calcular probabilidades son propias del paquete *pandas*. Este paquete al estar enfocado al análisis de datos ofrece una gran librería de comandos que facilitan el proceso.

Código 2.10: Práctica 8, ejercicio 1.1

```
def estimaHMM(sec, hid, alf, alf_hid):
    import pandas as pd


    if len(sec) != len(hid):
        print('sec y hid no tienen la misma longitud')


    pr = pd.Series(list(hid)).value_counts(normalize = True)
    mat_rec = pd.DataFrame(0, columns = alf_hid,index = alf_hid)
    for i in range(len(sec)-1):
        mat_rec.loc[hid[i],hid[i+1]] = mat_rec.loc[hid[i],hid[i+1]] +1


    total = []
    for i in range(len(alf_hid)):
        estado = alf_hid[i]
        total.append(hid.count(estado))

    T = pd.DataFrame(0, columns = alf_hid, index = alf_hid)

    for i in range(len(alf_hid)):
        estado = alf_hid[i]
        row = mat_rec.loc[estado,]

        freq = row.divide(sum(row))
        fila_T = T[estado:estado]
        T = T.replace(fila_T, freq)

    E = pd.DataFrame(0, columns = alf, index = alf_hid)

    for i in range(len(hid)):
        estado = hid[i]
        base = sec[i]
        E.loc[estado, base] = E.loc[estado,base] +1

    for i in range(len(alf_hid)):
        estado = alf_hid[i]
        row = E.loc[estado,]

        freq = row.divide(sum(row))
        E = E.replace(row, freq)


    return(pr,T,E)
```

2.2.6. Algortimo de Viterbi

Para estudiar el algoritmo de Viterbi, se ha escogido el ejercicio 3 de la práctica 8 donde se le pide al alumno que escriba un código para definir la función `ViterbiHMM`. Esta función va a dar la secuencia de estados ocultos a partir de su modelo de Markow oculto (`resHMM`) y su secuencia de estados visibles (`sec`).

Código 2.11: Práctica 8 , ejercicio 3.1 (código en R)

```
1  ViterbiHMM = function(sec,HMM,echo=FALSE)
2  { S = colnames(HMM$E); H = rownames(HMM$E);
3   p = HMM$p; T = HMM$T; E = HMM$E;
4   nH=length(H);
5
6   N = length(sec)
7   V = matrix(0,ncol=N,nrow=length(H)); rownames(V)=H; colnames(V)=sec;
8   M = matrix(0,ncol=N,nrow=length(H)); rownames(M)=H; colnames(M)=sec;
9
10  V[ ,1] = p * E[ ,sec[1]]; M[ ,1] = 1:nH
11
12  for ( iV in 2:N )
13  { prodMatr = V[ ,iV-1]*T
14     M[ , iV] = apply ( prodMatr, 2, which.max )
15     V[ , iV] = E [ ,sec[iV] ] * prodMatr[ cbind(M[,iV],1:nH) ]}
16
17  hid.sec = character(N)
18  cual = which.max(V[,N])
19  hid.sec[N] = H[cual]
20
21  for (iM in (N-1):1)
22  { cual = M[cual,iM+1]
23  hid.sec[iM] = H[cual] }
24  return(hid.sec)
25  }
```

Para poder obtener esa secuencia de estados ocultos, es necesario definir la matriz de Viterbi (`V`) y la matriz de punteros (`M`). Estas matrices se van a rellenar a partir de la información obtenida de la función `estimaHMM` (véase código 2.9), es decir, el argumento de la función llamado `HMM` . Lo esencial en la resolución de este ejercicio se encuentra en el manejo de las matrices, es decir, es muy importante saber cómo localizar una posición y rellenarla con un dato o extraer dicho dato para aplicarlo en otro punto. En R, véase el código 2.11, eso se hace usando corchetes y las coordenadas del dato de interés. Para localizar una posición dentro de la matriz pueden ser o bien ambas letras, números o uno de cada. A continuación se verá que en Python (código 2.12), el procedimiento es diferente.

Código 2.12: Práctica 8 , ejercicio 3.1

```
def ViterbiHMM(sec, resHMM):
    import pandas as pd
    pr = resHMM[0]
    T = resHMM[1]
    E = resHMM[2]

    alf_sec = E.columns
    alf_hid = E.index

    nH = len(alf_hid)

    V = pd.DataFrame(columns = list(sec), index = alf_hid)
    pntr = pd.DataFrame(columns = list(sec), index = alf_hid)

    V.iloc[:,0] = pr * E.loc[:,sec[0]]
    pntr.iloc[:,0] = alf_hid

    for i in range(1, len(sec)):
        prodMatr = V.iloc[:, i-1]*T
        max_values = prodMatr.max()

        for h in range(len(alf_hid)):
            V.iloc[h,i] = E.loc[alf_hid[h], sec[i]] * max_values.max()
            pntr.iloc[h,i] = max_values.idxmax()

    sec_hid = pd.DataFrame(columns = range(len(sec)), index = [0])
    sel_0 = V.iloc[:,len(sec)-1].apply(pd.to_numeric, errors = 'coerce')
    cual = sel_0.idxmax()
    sec_hid.iloc[0, len(sec)-1] = cual

    for hid in range(len(sec)-2, -1, -1):
        cual = pntr.ix[cual, hid+1]
        sec_hid.iloc[0,hid] = cual

    return(sec_hid)
```

En el caso de Python (código 2.12), como se ha comentado en el código 2.10, aquí también es necesario usar el paquete *pandas* para poder definir la función. Además, esto implica que todas las funciones usadas son propias de dicho paquete, como son `.idxmax()` o `.iloc()`.

Hablando de la localización de datos dentro de variables del tipo `pd.DataFrame`, los corchetes no sirven, como era en el caso de R. En este escenario, *pandas* ofrece tres funciones que se diferencian en función de la etiqueta que se está usando para localizar dicho dato (Tutorial - pandas, 2019):

- `.iloc[]`: se usa para seleccionar un dato dentro de una variable del tipo `pd.DataFrame` a través de su etiqueta (véase la línea 19 del código 2.12)

- `.loc[]`: se usa para seleccionar un dato dentro de una variable del tipo `pd.DataFrame` a través de su posición dentro de la matrix (véase la línea 23 del código 2.12)

- `.ix[]` : se usa para seleccionar un dato dentro de una variable del tipo `pd.DataFrame` a través de su posición y variable (véase la línea 32 del código 2.12)

Por otro lado, otras funciones características que se han usado en este código han sido `.idxmax()` y `.max()`. La primera sirve para obtener la etiqueta del valor máximo dentro de la variable y la segunda sirve para obtener el valor máximo persé de la variable. Cabe aclarar que en la línea 27 del código 2.12 se ha tenido que aplicar una serie de funciones para poder usar posteriormente `.idmax()`.

Debido al gran número de multiplicaciones cercanas al valor 0 que se realizan en la función `ViterbiHMM`, puede haber errores en la precisión de estos datos. Es por ello que aplicando logaritmos a esta función, se puede corregir este error y obtener resultados más precisos. Como se observa en el código 2.13, las principales modificaciones con respecto a la función `ViterbiHMM` se encuentran en el cálculo de las matrices `V` y `pntr` (véase desde la línea 9 hasta la línea 26 del código 2.13). En este caso, el paquete *pandas* no tiene una función que permita calcular el logaritmo, sin embargo el paquete *NumPy* tiene la función `np.log10()` que sí lo hace posible.

Código 2.13: Práctica 8 , ejercicio 4.1

```
def logViterbiHMM(sec, resHMM):
   import pandas as pd
   import numpy as np

   pr = resHMM[0]
   T = resHMM[1]
   E = resHMM[2]

   log_pr = np.log10(pr)
   log_T = np.log10(T)
   log_E = np.log10(E)
   alf_sec = E.columns
   alf_hid = E.index

   log_V = pd.DataFrame(columns = list(sec), index = alf_hid)
   pntr = pd.DataFrame(columns = list(sec), index = alf_hid)

   log_V.iloc[:,0] = log_pr + log_E.loc[:,sec[0]]
   pntr.iloc[:,0] = alf_hid

   for i in range(1, len(sec)):
       prodMatr = log_V.iloc[:, i-1]+log_T
       max_values = prodMatr.max()
       for h in range(len(alf_hid)):
           log_V.iloc[h,i] = log_E.loc[alf_hid[h], sec[i]] + max_values.
              max()
           pntr.iloc[h,i] = max_values.idxmax()

   sec_hid = pd.DataFrame(columns = range(len(sec)), index = [0])
   sel_0 = log_V.iloc[:,len(sec)-1].apply(pd.to_numeric, errors = 'coerce
      ')
   cual = sel_0.idxmax()
   sec_hid.iloc[0, len(sec)-1] = cual
   for hid in range(len(sec)-2, -1, -1):
       cual = pntr.ix[cual, hid+1]
       sec_hid.iloc[0,hid] = cual

   return(sec_hid)
```

2.2.7. Funciones y expresiones características de Python

Hay muchas expresiones y funciones que se encuentran en R y no en Python y viceversa. Comenzamos hablando de aquellas funciones propias de Python.

2.2.7.1. `Counter()`

El módulo *Counter* que se encuentra en Python permite contar los diferentes elementos que componen un grupo. El resultado que se obtiene es una variable del tipo diccionario donde se dispone el elemento y cuántas veces se encuentra.

Como ejemplo se ha cogido el ejercicio 2.1 de la práctica 6 (véase el código 2.14 donde se le pide al alumno que defina una función que le permita obtener el modelo multinomial de una secuencia de aminoácidos (`old_aa`).

Código 2.14: Práctica 6, ejercicio 2.1

```
1  def ModeloMultinomial_general(seq):
2      from collections import Counter
3      count = Counter(seq)
4      num = list(count.values())
5      keys = list(count.keys())
6      freq = []
7      for i in range(0,len(num)):
8          f = num[i]
9          freq.append(f/len(seq))
10
11     mm = dict(zip(keys,freq))
12     return(mm)
13
14 ModeloMultinomial_general(old_aa)
15
16 {'K': 0.06863903633747248,
17  'V': 0.03733572851247939,
18  'Y': 0.06529880971667636,
19  'S': 0.06615649986181658,
20  'L': 0.09935387342399436,
21  'F': 0.07475007862159663,
22  'I': 0.13930317440652606,
23  'P': 0.019119342818751012,
24  'D': 0.02792734411481612,
25  'A': 0.01458073246738395,
26  'T': 0.03864132351119286,
27  'E': 0.021413663957001134,
28  'N': 0.07522419067404916,
29  '*': 0.08063955095155957,
30  ...
31  'X': 0.00014771330277415113}
```

Esta función facilita mucho el recuento de secuencias aminoacídicas ya que evitamos definir una función `count()` para cada uno de los aminoácidos. Sin embargo, acceder a los datos que se obtiene al usar `Counter()` es complicado. Si ahora quisiéramos usar todos los resultados, tendríamos que pasar este tipo a variable tipo lista y se perdería información. Además, el resultado no es un diccionario normal, sino que es un tipo de diccionario especial del módulo de `Counter()`. Así hay funciones que se pueden aplicar a un diccionario, pero que no se pueden aplicar a un objeto `Counter`. Por ello, se recomienda su uso para hacer recuento de elementos y visualización de datos, no para manipularlos después.

2.2.7.2. *List comprehensions*

Las *list comprehensions* son una forma de sintaxis ampliamente usada en Python cuando se quiere iterar sobre elementos y recoger los resultados en una variable de tipo lista.

Para ejemplificarlo se ha cogido un fragmento del código realizado para el ejercicio 4 de la práctica 3 (código 2.15). En este ejercicio, se le pide al alumno que busque las posiciones de los codones 'GAU' en la variable `j_matrix_rna` y las guarde en una nueva variable, `indices`.

Código 2.15: Práctica 4, ejercicio 3

```
1 indices = [i for i,x in enumerate(j_matrix_rna) if x=='GAU']
```

Las *list comprehensions* permiten escribir el código de una forma más concisa, útil y con una ejecución más rápida. Este tipo de sintaxis sería el equivalente al comando `apply()` de R. Sin embargo, `apply()` tiene un argumento que permite aplicar la función por filas o por columnas a la matriz, operación que no se puede realizar de forma tan sencilla en Python. Para poder hacerlo, previamente se tienen que disponer los elementos en el orden que queremos que se aplique la función.

2.2.7.3. `''.join()`

En Python no existe el comando `c2s()` que sí se encuentra en R. Este comando permite pasar una secuencia de elementos que se encuentra 'A' 'B' 'C' 'D' 'E' a 'ABCDE'. Su equivalente en Python sería `''.join()`. Esta función solamente es aplicable a variables del tipo string.

Como ejemplo se ha cogido la primera línea del código escrito para el ejercicio 3 de la práctica 4 (véase el código 2.16)

Código 2.16: Práctica 4, ejercicio 3, línea 1

```
1 j_matrix_rna = [''.join(matrix_rna[i]) for i in range(len(matrix_rna))]
```

2.2.7.4. `np.sample()`

Python tiene un módulo llamado `random` que se usa para hacer secuencias aleatorias a partir de un set de elementos. Sin embargo, no permite indicarle la probabilidad con la que pueden ser seleccionados estos elementos, argumento que sí aparece en el comando `sample()` en R. La función que más se asemeja es `np.sample()` y se encuentra dentro del paquete *NumPy*. Los argumentos que tiene `np.sample()` son los mismos que aparecen en `sample()`. Este comando se ha usado en el ejercicio 1.3 de la práctica 6 (código 2.17). En este ejercicio se le pide al alumno que cree una secuencia aleatoria de DNA definida por un modelo multinomial previamente calculado.

Código 2.17: Práctica 6, ejercicio 1.3

```
1 import numpy as np
2
3 words = ['A','C','G','T']
4 sample = np.random.choice(words, size = len(old_dna), replace = True,
5                           p = [0.37,0.12,0.13,0.38])
```

Como se observa, las probabilidades (`p`) y los elementos (`words`) se pueden proporcionar en formato lista y no en formato `np.array`. Además, tiene otro argumento, `replace`, mediante el que se indica si se pueden repetir los elementos en la secuencia aleatoria. Esta opción también se encuentra en `sample()` de R. No obstante, la variable que se obtiene es del tipo `np.random.choice`, por lo que, al igual que en la subsección 2.2.7.1, la variable obtenida es recomendable que sólo se use para su visualización.

Capítulo 3

Conclusión

La resolución de los ejercicios propuestos en Python y en R se asemejan mucho. Sin embargo, hay ciertas diferencias que hacen que la sintaxis del código sea más complicada en Python que en R.

La lectura de archivos fasta en Python no es muy compleja, solo hay que usar las funciones `SeqIO.read()` o `SeqIO.parse()` dependiendo del número de secuencias que componen el archivo. La naturaleza de la variable donde se guarda la información permite ejecutar funciones y atributos de manera sencilla. Además, la secuencia en sí actúa como si fuese una variable de tipo string, por lo que su manipulación no es complicada.

La librería de variables disponibles en cada lenguaje ha sido la mayor diferencia que se ha encontrado en el desarrollo de este trabajo (Tabla 2.1). En R, disponer de matrices y factores facilita la disposición de los datos y, por lo tanto, su análisis. Por otro lado, en Python se tiene que recurrir a paquetes externos (*pandas* y *NumPy*) para poder tener este tipo de variables que se rigen por funciones propias. Esto complica la realización de las prácticas. Además, la naturaleza de estas variables es diferente a las que vienen definidas en Python, provocando en muchas ocasiones problemas de compatibilidad. En este aspecto, R es más sencillo y adecuado para alumnos que están empezando a conocer el lenguaje de programación.

La representación gráfica de datos es más intuitiva en Python. Gracias al paquete *Matplotlib*, se pueden representar gráficos muy visuales de manera sencilla. Además, cuenta con una amplia librería de gráficos que se adaptan al tipo de datos y a la información que se quiere mostrar. Su recurso web es muy amplio (Tutorials - Matplotlib, 2019), donde encontramos todos estos tipos de gráficos acompañados de tutoriales muy sencillos y detalladamente explicados.

Hablando del alineamiento de secuencias, en ambos lenguajes el procedimiento es muy similar, permitiendo que esta tarea se haga de manera fácil y rápida. Biostrings y Biopython cuentan con una librería de matrices de substitución, haciendo que no sea necesario descargar la matriz de la web, importarla en el código y modificarla para que se ajuste al formato que se pida en cada lenguaje. Por un lado, en R, la función `alignPairwise` cuenta con distintos argumentos, como son `type`, `gapOpening` y `gapExtension`, que hacen posible definir con más exactitud el alineamiento que se busca. Por otro lado, en Python, la función `AlignPairwise.Aligner()` aplica el algoritmo de alineamiento adecuado en

función de las secuencias que queremos alinear. Asimismo, el hecho de tener una variable donde se almacene de forma automática toda la información sobre el alineamiento hace que el acceso a los resultados sea mucho más sencillo.

En el caso del estudio de las cadenas de Markov ocultas y el algoritmo de Viterbi, es necesario el uso de los paquetes *pandas* y *NumPy* para la definición de las matrices, en este caso `pd.DataFrame`, su modificación y para el cálculo de logaritmos. Esto implica que las funciones que se usen a la hora de variar dichas matrices van a ser propias de estos paquetes y no las que se encuentran definidas por defecto en Python. En R, trabajar con matrices es más sencillo que en Python, con el uso de corchetes se puede localizar la coordenada de la matriz que se quiera. Sin embargo, en Python hay que tener especial cuidado porque dependiendo de la naturaleza de dichas coordenadas, es decir si usamos la coordenadas en sí o las etiquetas que hayamos usado en las columnas y filas, se tiene que usar una función diferente (`.iloc[ ]`, `.loc[ ]` o `.ix[ ]`).

Por último, Python cuenta con funciones y expresiones propias que facilitan la sintaxis del código. El módulo `Counter` es de gran utilidad cuando se quiere hacer un recuento de aminoácidos de una secuencia. También, las *list comprehensions* facilitan la aplicación de una función a todos los elementos de una variable y guarda los resultados en otra variable de tipo lista. Es el equivalente a la función `apply()` de R, pero sin ese argumento que permite aplicar la función por filas o por columnas cuando los datos se encuentran en formato matriz.

En conclusión, los lenguajes Python y R se asemejan mucho salvo algunos matices. Se pueden obtener los mismos resultados a través de un código muy semejante. Sin embargo, para alumnos que están aprendiendo a programar por primera vez, el lenguaje más apropiado es R. Los tipos de variables que tiene y sus funciones propias facilitan la obtención de los resultados pedidos en las prácticas. Python es un lenguaje para aquellos que quieran ampliar sus conocimientos puesto que, teniendo unas nociones básicas de R, programar en Python se hace más sencillo.

3.1. Futuras aplicaciones del trabajo

La realización de estos códigos se ha hecho desde el punto de vista del alumnado. Esto quiere decir que se ha partido de unas nociones básicas de Python para escribir estos códigos de manera sencilla y de fácil compresión.

En el caso de que se quiera introducir el lenguaje Python en la asignatura Informática Aplicada a la Bioquímica, habría que cambiar algunos aspectos claves para que el alumno pueda realizar y comprender las prácticas.

- Para comenzar, se necesita ver Biopython y como se leen los ficheros fasta usando `SeqIO.read()` y `SeqIO.parse()` porque son necesarias para comenzar cada práctica que se haga. Además, se requiere de unas nociones básicas de las variables `SeqRecord` y `Seq`. Estas van a ser los tipos de variables con los que se van a empezar a trabajar y de donde se va a partir en todas las prácticas que se realicen.

- Posteriormente, es recomendable el estudio de los paquetes *NumPy* y *pandas* y sus funciones básicas (ej. su naturaleza, cómo se definen, como modificar los datos, etc.). Estos paquetes cuentan con un recurso web muy completo donde se encuentran todas las funciones y ejemplos explicados de cada una (Tutorial - pandas, 2019)(Tutorial - NumPy, 2019).

- Otro aspecto que se debería cambiar sería la introducción de los bucles *for* y *while*. Ambos bucles deberían explicarse en las primeras unidades, ya que en la práctica 3 se le pide al alumno que aplique una función usando `apply()` a todos los elementos de una variable. En Python esto se tiene que hacer usando *list comprehension* donde se usan los bucles *for*.

- La representación gráfica de datos no es necesaria explicarla antes. Sin embargo, se requiere de una explicación previa de *matplotlib* y sus nociones básicas. Para esto, el recurso web del paquete (Tutorials - Matplotlib, 2019) cuenta con muchos tutoriales que pueden servir de apoyo.

Anexo A

Tutoriales de instalación de programas y paquetes usados

A.1. Spyder (Anaconda Python/R Distribution, 2019)

- Descargamos e instalamos el paquete de Anaconda que se encuentra disponible en su página web (Anaconda Python/R Distribution, 2019) (figuras A.1 y A.2).

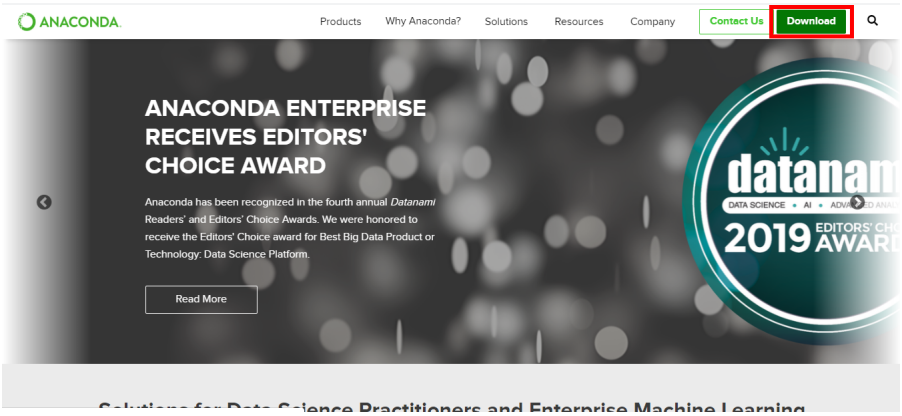


Figura A.1: Descarga de Spyder.

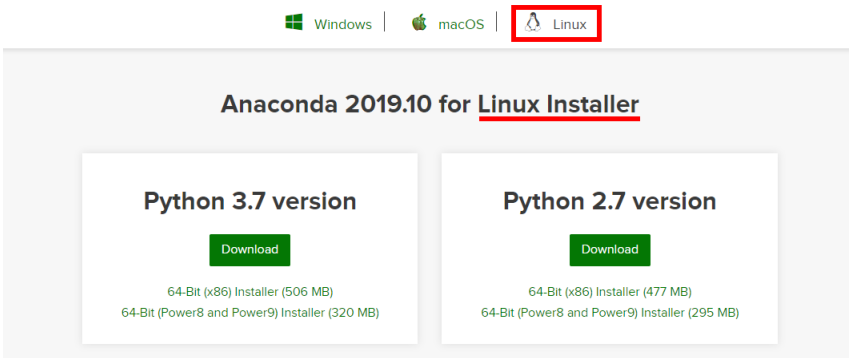


Figura A.2: Selección de Sistema Operativo.

- Una vez se haya completado la instalación, abrimos la terminal y escribimos el comando `spyder` (figura A.3).

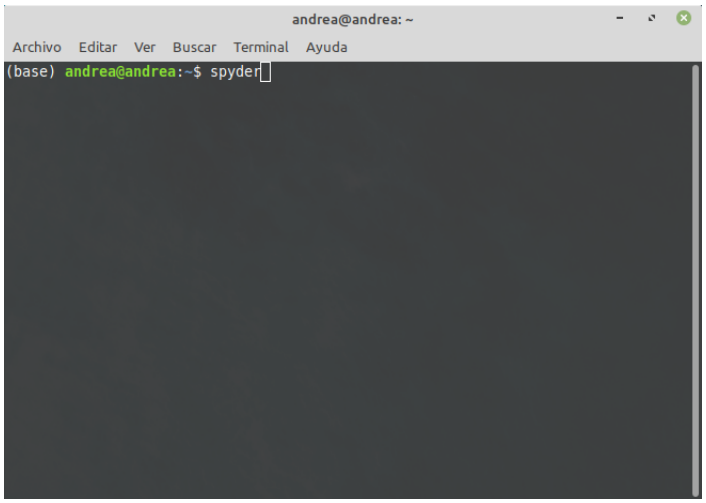


Figura A.3: Llamada a Spyder desde la terminal.

- Tras unos segundos, el IDE se cargará y se iniciará (figura A.4).

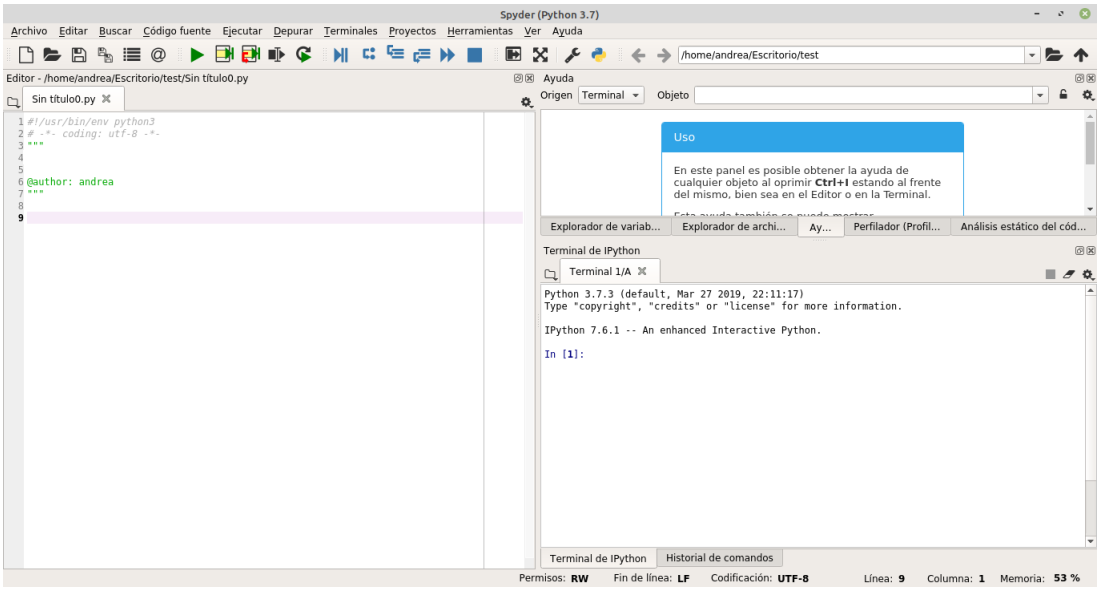


Figura A.4: IDE de Spyder.

A.2. Biopython (Biopython: Anaconda Cloud, 2019)

Como estamos trabajando con un entorno de desarrollo de Anaconda, la instalación del paquete se tiene que hacer a través de comandos `conda`. Para instalar Biopython abrimos la terminal y escribimos uno de los siguientes comandos:

Código A.1: Instalación Biopython

```
1 conda install -c conda-forge biopython
2 conda install -c conda-forge/label/gcc7 biopython
3 conda install -c conda-forge/label/cf201901 biopython
```

Una vez que la instalación haya concluido, ya tenemos el paquete instalado en Spyder. No es necesario cargar el paquete entero cada vez que queramos usarlo, sino que se importarán las herramientas que sean necesarias.

A.3. Instalación de pandas, NumPy y matplotlib (Pandas: Anaconda Cloud, 2019) (Numpy: Anaconda Cloud, 2019) (Matplotlib: Anaconda Cloud, 2019)

Al igual que en el caso anterior, la instalación la realizamos a través de `conda`. Abrimos la terminal y escribimos los siguientes comandos:

Código A.2: Instalación pandas, NumPy y matplotlib

```
1 conda install -c anaconda pandas
2 conda install -c anaconda numpy
3 conda install -c conda-forge matplotlib
```

Una vez instalados, cada vez que se quieran usar, se tiene que importar de la siguiente manera:

Código A.3: Importar pandas, NumPy y matplotlib

```
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
```

Bibliografía

Abdurakhmonov, I. Y. (2016). *Bioinformatics: Basics, Development and Future*, chapter 1, pages 1–13. Intech Open. Chapter of *Bioinformatics - Updated Features and Applications*.

Anaconda (2019). `https://www.anaconda.com/`. Accedido: Diciembre 2019.

Anaconda Python/R Distribution (2019). `https://www.anaconda.com/distribution/`. Accedido: Diciembre 2019.

Andrea Morales (2019). Informática Aplicada a la Bioquímica en Python - Códigos. `https://github.com/andreammo/Informatica-Aplicada-a-la-Bioquimica-en-Python/tree/master`. Accedido: Diciembre 2019.

Bassi, S. (2007). A Primer on Python for Life Science Researchers. *PLoS Computational Biology*, 3(11):2052–2057.

Biopython: Anaconda Cloud (2019). `https://anaconda.org/conda-forge/biopython`. Accedido: Diciembre 2019.

Chang, J., Chapman, B., Friedberg, I., Hamelryck, T., de Hoon, M., Cock, P., Wilczyński, B., Antao, T., and Talevich, E. (2019). Biopython 1.74 Tutorial and Cookbook. 2019(July):339.

Cock, P. J., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., Friedberg, I., Hamelryck, T., Kauff, F., Wilczynski, B., and De Hoon, M. J. (2009). Biopython: Freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423.

Cotton, R. (2013). *Learning R.* O'Reilly Media Inc.

Ekmekci, B., McAnany, C., and Mura, C. (2016). An Introduction to Programming for Bioscientists: A Python-Based Primer. *PLoS Computational Biology*, 12(6):1–43.

GenBank (2019). `https://www.ncbi.nlm.nih.gov/genbank/`. Accedido: Diciembre 2019.

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science and Engineering*, (9):90–95.

Integrated Development Environments, IDE: in Python (2019). `https://wiki.python.org/moin/IntegratedDevelopmentEnvironments`. Accedido: Diciembre 2019.

Komodo (2019). https://www.activestate.com/products/komodo-ide/python-editor/. Accedido: Diciembre 2019.

Linux Mint (2019). https://linuxmint.com/. Accedido: Diciembre 2019.

Lutz, M. (2013). *Learning Python 5th edition*. O'Reilly Media Inc.

Madeira, F., Park, Y., Lee, J., Buso, N., Gur, T., Madhusoodanan, N., Basutkar, P., Tivey, A., Potter, S., Finn, R., and Lopez, R. (2019). The EMBL-EBI search and sequence analysis tools APIs in 2019. https://www.embl.org/. Accedido: Diciembre 2019.

Matplotlib: Anaconda Cloud (2019). https://anaconda.org/conda-forge/matplotlib. Accedido: Diciembre 2019.

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 51–56. SciPy.

National Center for Biotechnology Information (NCBI) (1988). https://www.ncbi.nlm.nih.gov/. Accedido: Diciembre 2019.

Numpy: Anaconda Cloud (2019). https://anaconda.org/anaconda/numpy. Accedido: Diciembre 2019.

Package, S., View, M., and Bairoch, A. (2000). Bioinformatics: companies are selling research software. *Nature Biotechnology*, 18:31–34.

PairwiseAligment Function (2019). https://www.rdocumentation.org/packages/Biostrings/versions/2.40.2/topics/pairwiseAlignment. Accedido: Diciembre 2019.

Pandas: Anaconda Cloud (2019). https://anaconda.org/anaconda/pandas. Accedido: Diciembre 2019.

Perkel, J. M. (2015). Programming: Pick up Python. *Nature*, 518(7537):125–126.

PyCharm (2019). https://www.jetbrains.com/es-es/pycharm/. Accedido: Diciembre 2019.

Spyder (2019). https://www.spyder-ide.org/. Accedido: Diciembre 2019.

Tutorial - NumPy (2019). https://numpy.org/devdocs/user/quickstart.html. Accedido: Diciembre 2019.

Tutorial - pandas (2019). https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html. Accedido: Diciembre 2019.

Tutorials - Matplotlib (2019). https://matplotlib.org/tutorials/index.html. Accedido: Diciembre 2019.

Van Der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30.