Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Estudio de Complejidad en Secuencias de Aminoácidos de Proteínas

Tesis de Licenciatura en Ciencias de la Computación

Mariano Oca

Director: Pablo Turjanski
Codirector: Ignacio Sánchez
Buenos Aires, 2025

ESTUDIO DE COMPLEJIDAD EN SECUENCIAS DE AMINOÁCIDOS DE PROTEÍNAS

El estudio de complejidad de aminoácidos nos remonta a los orígenes de la vida misma. Secuencias de ADN se traducen en secuencias de aminoácidos, que se ven completamente aleatorias. Sin embargo, una complejidad intrínseca a estas secuencias exhala vida, estas son la base para producir complejas estructuras indispensables para la vida como la conocemos. Sin embargo, carecemos de métodos sencillos que nos permitan diferenciarlas de secuencias estériles de aminoácidos que no producen nada. Esto implica, que ante nosotros tenemos la Biblioteca de Alejandría escrita en un idioma al que somos ajenos, que existe un orden y un camino donde sólo podemos ver caos. Esta es la motivación de este este trabajo, la búsqueda de ese camino, encontrar alguna métrica de complejidad que nos permita discernir entre secuencias de aminoácidos que codifican proteínas funcionales de las que no. Con el objetivo de sentar un precedente para cualquier interesado en esta materia y proveer una herramienta sencilla y elegante para el estudio de las secuencias que rigen nuestras vidas.

Palabras claves: Complejidad, Secuencias de Aminoácidos, Proteínas, Icalc, Discrepancia, Bennett, Kolmogorov, Entropía.

A mi papá.

Índice general

1.. Introducción 1
1.1. Contexto y Relevancia 1
1.2. Azar y Complejidad 1
1.3. Objetivos 2

2.. La Herramienta y Metodología 3
2.1. Generación de Datasets de Control 5
2.2. Más Sobre Cómo Funciona la Herramienta 8

3.. Métricas de Complejidad y Sus Resultados 11
3.1. Icalc 11
3.2. Discrepancia 13
3.3. Discrepancia en Bloque 16
3.4. Métricas provistas por OACC 19
3.4.1. Kolmogorov - BDM algorithmic complexity estimation (bits) 20
3.4.2. Bennnett - BDM logical depth estimation (steps) 21
3.4.3. Shannon entropy (bit(s)) 23
3.4.4. Second order entropy (bit(s)) 24
3.4.5. Compression length using gzip (bits) 26

4.. Conclusiones y Trabajo Futuro 29
4.1. Trabajo Futuro 30

5.. Anexo: Documentación de la Herramienta 31
5.1. Requerimientos y Versiones 31
5.2. Convenciones 31
5.3. Estructura del proyecto 31
5.4. Documentación de Funciones 32

1. INTRODUCCIÓN

1.1. Contexto y Relevancia

Las moléculas de proteínas se pueden describir como secuencias finitas compuestas de unos elementos denominados aminoácidos. Generalmente se pueden observar en estas secuencias (también llamadas polipéptidos) 20 tipos de aminoácidos distintos, aunque en realidad se componen de muchos tipos más pero que son poco probables de observar. Dentro del ámbito de la dinámica molecular existe un hecho sumamente intrigante: la mayoría de las secuencias conocidas de aminoácidos que se encuentran en la naturaleza parecen indistinguibles de secuencias de aminoácidos generadas al azar. Sin embargo, si se sintetiza en un laboratorio un polipéptido ubicando sus aminoácidos de manera aleatoria en la secuencia, es altamente probable que esta cadena no se comporte como una proteína, es decir, no se va a plegar a estructuras específicas ni va a funcionar en un contexto celular [1, 2]. Ante esta situación, es de interés buscar "códigos estructurales" en secuencias de proteínas, considerando qué relaciones aparecen entre los patrones de grupos de aminoácidos. Sin embargo, se trata de una tarea que se vuelve combinatoriamente prohibitiva para analizar exhaustivamente en todas las secuencias de proteínas. Una posible manera de aproximarse a la respuesta sería reformulando la pregunta a si las proteínas que se encuentran en la naturaleza son realmente indistinguibles de secuencias de aminoácidos generadas al azar o simplemente no contamos con una función que permitan distinguirlas. Para ello, nos basaremos en conceptos ligados al azar y a la complejidad.

1.2. Azar y Complejidad

El concepto de azar y complejidad están intrínsecamente ligados, la complejidad depende naturalmente del contexto y de la información disponible para los agentes, ahí es donde entra el concepto de azar. Porque, en términos de información esperada, un mensaje que indique que algo que tiene 100 % de probabilidad va a pasar, carece de información, ergo carece de complejidad. Siguiendo esa idea, se esperaría que la información de que algo poco probable vaya a pasar tiene más complejidad que la tuviera si dicho hecho fuera más probable. Puntualmente, en el caso de un *stream* binario, la certeza de que el próximo bit es 1 es más relevante si la probabilidad de esto suceda es 30 % a que sea 70 %. Durante la historia de la computación no faltaron los esfuerzos por desambiguar este concepto. A continuación veremos algunos hitos en este proceso.

Una medida de complejidad para los elementos de un conjunto es una función que asigna a cada elemento del conjunto un número. Se espera que la mayoría de los elementos sean muy complejos. A mediados de 1960 Kolmogorov [3] definió una medida de complejidad de palabras, hoy conocida como la complejidad de Kolmogorov, que asocia a las palabras de máxima complejidad con la noción de que carecen de regularidad, es decir, con que carecen de un patrón de formación.

A mediados de los años 70's Gregory Chaitin dio una variante de la función de complejidad de Kolmogorov, siendo esta una medida de cantidad de información, con propiedades idénticas a la noción de entropía de la información de Shannon [4]. Con esta variante, Chaitin introduce una definición de aleatoriedad para palabras infinitas: una palabra es

aleatoria si todos sus segmentos iniciales tienen máxima complejidad. Esto es lo mismo que decir que todos los segmentos iniciales tienen máxima información. También es equivalente a decir que la única manera de describir a los segmentos iniciales de una secuencia aleatoria es, esencialmente, describiéndolos explícitamente.

Si bien la complejidad de Chaitin-Kolmogorov es una poderosa herramienta teórica, dado que no es computable, es imposible utilizarla en aplicaciones prácticas. Se han dado varias definiciones alternativas de complejidad de palabras, en general son longitudes de descripciones minimales para distintos métodos de descripción.

En los años 70's Lempel y Ziv publicaron una medida de complejidad eficientemente computable, que combina un método procedural con propiedades combinatorias de las palabras. Se la conoce como la complejidad de Lempel-Ziv [5]. La complejidad de Lempel-Ziv es una medida computable muy eficientemente mediante un algoritmo que ejecuta en tiempo y memoria lineal respecto del tamaño de la entrada. El inconveniente de esta medida es de índole teórica: hay infinitas palabras cuya complejidad de Lempel-Ziv es menor que cualquier valor dado; esto es contrario a lo que ocurre en la complejidad de Chaitin-Kolmogorov. Por otro lado, la definición de Lempel-Ziv combina propiedades combinatorias de las palabras con un tratamiento puramente procedural, de forma tal que es difícil demostrar sus propiedades.

La medida de complejidad presentada en el trabajo de Becher y Heiber[6] es una medida de complejidad de palabra basada en conteo de repeticiones. Las palabras más complejas son las secuencias de Bruijn, las más simples son las secuencias de una sola racha, y la mayoría de las secuencias tienen alta complejidad.

1.3. Objetivos

El objetivo general de la tesis es intentar responder la pregunta de si las proteínas que se encuentran en la naturaleza son realmente indistinguibles de secuencias de aminoácidos generadas al azar. Para ello se intentarán explorar distintas medidas de complejidad y se procederá a compararlas aplicándolas sobre distintos conjuntos de datos. Este objetivo, se desglosa en los siguientes objetivos específicos:

1. Generar las fuentes de datos de secuencias de aminoácidos sobre las cuales se van a correr las distintas funciones de complejidad. Las tareas asociadas a este objetivo incluyen

   i) obtener secuencias de aminoácidos naturales de fuentes de datos públicas[1];

   ii) generar secuencias aleatorias tanto con distribución uniforme de aminoácidos como manteniendo la distribución natural de ellos.

2. Implementar distintas funciones de complejidad para que puedan ser ejecutadas tomando como entrada secuencias de aminoácidos y analizar los resultados obtenidos al aplicarlas en las fuentes de datos del punto anterior.

3. Desarrollar una herramienta que permita el cálculo y el análisis de diferentes funciones de complejidad aplicadas a fuentes de datos de secuencias de aminoácidos.

[1] **UniProt**. Se utilizará *Reviewed (Swiss-Prot)* (www.uniprot.org/help/downloads)

2. LA HERRAMIENTA Y METODOLOGÍA

La herramienta[7] cuenta de una serie de funciones en un proyecto en `Python` que son sumamente útiles a la hora de estudiar complejidad en secuencias de aminoácidos. Es escogido este lenguaje por su amplio uso y versatilidad indistintamente del sistema operativo, además resulta mucho más fácil escribir código legible, mantenible y escalable en `Python` que en otros lenguajes (como `C` por ejemplo). Incluso, cuenta con una amplia variedad de bibliotecas que permiten el uso de otros programas, puntualmente en este trabajo se hace uso de la biblioteca `rpy2` para ejecutar una métrica de complejidad escrita en `R`.

Como se explicó en el apartado de objetivos, trabajaremos con el dataset *Reviewed (Swiss-Prot)* de **UniProt**. Esto consiste en un único dataset en forma del archivo `uniprot_sprot.fasta`. Este es un archivo multifasta, es decir, tiene varias entradas del tipo:

```
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R OS=Frog virus 3
(isolate Goorha) OX=654924 GN=FV3-001R PE=4 SV=1
MAFSAEDVLKEYDRRRRMEALLLSLYYPNDRKLLDYKEWSPPRVQVECPKAPVEWNNPPSEKGLIVGHFSGIKY
KGEKAQASEVDVNKMCCWVSKFKDAMRRYQGIQTCKIPGKVLSDLDAKIKAYNLTVEGVEGFVRYSRVTKQHVA
AFLKELRHSKQYENVNLIHYILTDKRVDIQHLEKDLVKDFKALVESAHRMRQGHMINVKYILYQLLKKHGHGPD
GPDILTVKTGSKGVLYDDSFRKIYTDLGWKFTPL
>sp|Q6GZX3|002L_FRG3G Uncharacterized protein 002L OS=Frog virus 3 (isolate
Goorha) OX=654924 GN=FV3-002L PE=4 SV=1
MSIIGATRLQNDKSDTYSAGPCYAGGCSAFTPRGTCGKDWDLGEQTCASGFCTSQPLCARIKKTQVCGLRYSSK
GKDPLVSAEWDSRGAPYVRCTYDADLIDTQAQVDQFVSMFGESPSLAERYCMRGVKNTAGELVSRVSSDADPAG
GWCRKWYSAHRGPDQDAALGSFCIKNPGAADCKCINRASDPVYQKVKTLHAYPDQCWYVPCAADVGELKMGTQR
DTPTNCPTQVCQIVFNMLDDGSVTMDDVKNTINCDFSKYVPPPPPPKPTPPTPPTPPTPPTPPTPPTPPTPRPV
HNRKVMFFVAGAVLVAILISTVRW
>sp|Q197F8|002R_IIV3 Uncharacterized protein 002R OS=Invertebrate
iridescent virus 3 OX=345201 GN=IIV3-002R PE=4 SV=1
MASNTVSAQGGSNRPVRDFSNIQDVAQFLLFDPIWNEQPGSIVPWKMNREQALAERYPELQTSEPSEDYSGPVE
SLELLPLEIKLDIMQYLSWEQISWCKHPWLWTRWYKDNVVRVSAITFEDFQREYAFPEKIQEIHFTDTRAEEIK
AILETTPNVTRLVIRRIDDMNYNTHGDLGLDDLEFLTHLMVEDACGFTDFWAPSLTHLTIKNLDMHPRWFGPVM
DGIKSMQSTLKYLYIFETYGVNKPFVQWCTDNIETFYCTNSYRYENVPRPIYVWVLFQEDEWHGYRVEDNKFHR
RYMYSTILHKRDTDWVENNPLKTPAQVEMYKFLLRISQLNRDGTGYESDSDPENEHFDDESFSSGEEDSSDEDD
PTWAPDSDDSDWETETEEEPSVAARILEKGKLTITNLMKSLGFKPKPKKIQSIDRYFCSLDSNYNSEDEDFEYD
SDSEDDDSDSEDDC
...
```

Donde cada entrada de este archivo tiene dos partes. La primera, que consiste en un header que comienza con el carácter “>” que consta de una descripción de la secuencia, y luego, en una o más líneas la representación de una única secuencia de aminoácidos donde estos se representan como letras, siguiendo el esquema denotado por la tabla 2.1 [1].

[1] Una muy buena explicación de este formato puede encontrarse en (es.wikipedia.org/wiki/Formato_FASTA)

Código de Aminoácido	Significado	Código de Aminoácido	Significado
A	Alanina	M	Metionina
C	Cisteína	N	Asparagina
D	Ácido aspártico	P	Prolina
E	Ácido glutámico	Q	Glutamina
F	Fenilalanina	R	Arginina
G	Glicina	S	Serina
H	Histidina	T	Treonina
I	Isoleucina	V	Valina
K	Lisina	W	Triptófano
L	Leucina	Y	Tirosina

Tab. 2.1: Esquema FASTA

En la figura 2.1 veremos los datos descargados antes (`uniprot_sprot`, `usp`) y después (`uniprot_sprot` *filtered*, `usp_f`) del filtrado que consiste en eliminar las secuencias de largo menor a 50 aminoácidos. Estas son 12.839 de las 571.864 secuencias totales. También ordenamos el dataset de menor a mayor en función del largo de la secuencia para poder trabajar con los datos más cómodamente.

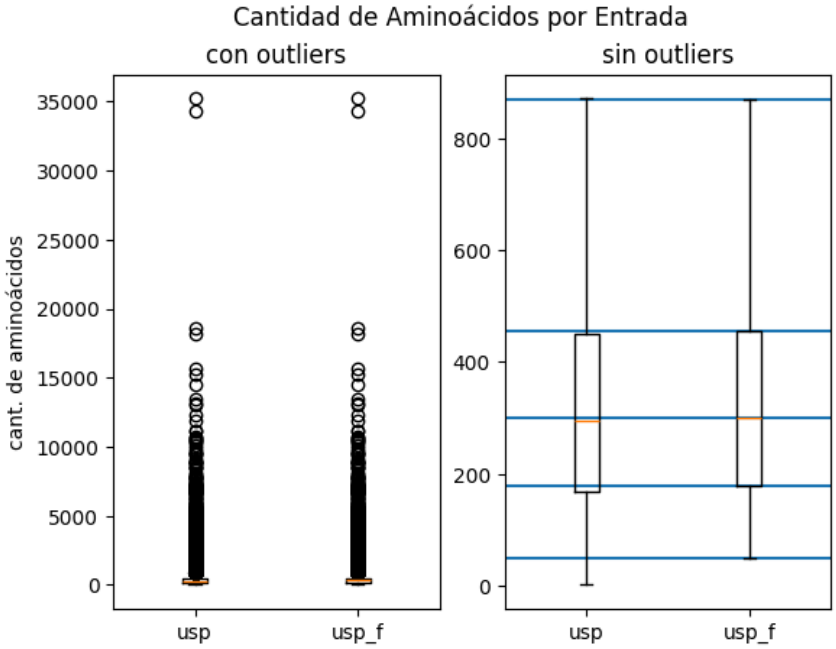

Fig. 2.1: Distribución de las entradas del dataset original (`usp`) comparado con la distribución después de remover las entradas de largo menor a 50 (`usp_f`).

Para poder hacer esto, utilizamos de la herramienta `filter_to_file(in_file:`*`str`*`, out_file:`*`str`*`, criteria:`*`seq_record -> Bool`*`) -> `*`None`*, que permite dado cualquier dataset (en formato multifasta), filtrarlo según algún criterio. Donde `in_file` consiste en el archivo desde donde se leerá la entrada, `out_file` consiste en el archivo donde se guardará el dataset ya filtrado y `criteria` es una función que consiste en el criterio de filtrado que

se aplicará a cada secuencia, si da `True` la secuencia se guarda, si no, se descarta. Esta función al igual que todas las funciones en esta herramienta no modifica los archivos que lee. Entonces, el filtrado antes descrito es fácilmente realizable por la herramienta con el código a continuación.

```
filter_to_file("usp.fasta", "usp_f.fasta",
               lambda seq_record : len(seq_record) >= 50)
```

Luego para ordenarlo, `sort_to_file(in_file:str, out_file:str) -> None`, donde `in_file` y `out_file` siguen el mismo esquema que en la función `filter_to_file` tenemos que el resultado de esta función es guardar en el archivo `out_file` una copia ordenada de menor a mayor en largo de secuencia del dataset de entrada. Entonces, para ordenar `usp_f.fasta` en función del largo de las entradas en forma ascendente, sobrescribiendo el mismo archivo.

```
sort_to_file("usp_f.fasta", "usp_f.fasta")
```

2.1. Generación de Datasets de Control

Esta herramienta nos permite, además, generar datasets de control. Los casos por defecto son *shuffled* y *random*, aunque otros pueden agregarse fácilmente. Estos dos dataset nos permitirán tener contra qué comparar el resultado de complejidad obtenido. Se generaron 10 *datasets random* y 10 *datasets shuffled*, todos de forma pseudoaleatoria. Para garantizar la replicabilidad del experimento, se utilizaron semillas aleatorias del 1 al 10. De este modo, el primer *dataset random* (`usp_f_r01.fasta`) y el primer *dataset shuffled* (`usp_f_s01.fasta`) comparten la semilla 1; el segundo par (`usp_f_r02.fasta` y `usp_f_s02.fasta`), la semilla 2; y así sucesivamente hasta el décimo (`usp_f_r10.fasta` y `usp_f_s10.fasta`). Estas semillas no poseen ninguna característica especial; simplemente se utilizan para evitar que una elección aleatoria que produzca una distribución atípica sesgue los resultados, lo cual podría conducir a conclusiones erróneas.

Para las secuencias *shuffle* se aplicó a la función `shuffle()` (provista por la biblioteca `random` de `Python`) a la secuencia de cada entrada en el archivo .fasta original, esto se guardó en un nuevo archivo `.fasta.` Para generar el dataset *random* se tomaron de cada entrada del archivo original el largo de la secuencia y se generaron una secuencia del mismo largo pero con distribución aleatoria uniforme sobre el alfabeto de los posibles aminoácidos (`"ACDEFGHIKLMNPQRSTVWY"`), el header se mantuvo igual. A continuación veremos un ejemplo de esto.

La función `shuffle_to_file(in_file:str, out_file:str, seed = 1) -> None`, donde `in_file` y `out_file` siguen el mismo esquema que en la función `filter_to_file`, notemos que `seed` es por defecto 1 si no se instancia el parámetro, el rango debe ser de 1 a 99 inclusive. Con semilla aleatoria = 1, tenemos que el resultado de esta función de aplicarla a un archivo que contenga las secuencias anteriormente mostradas es la siguiente:

```
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R OS=Frog virus 3
(isolate Goorha) OX=654924 GN=FV3-001R PE=4 SV=1
KPGLRQGGDFIYAGMKSEIREEPMVYAYLDIDWKLMIQQISTNCVYKKGWKWRVDKGWDADKDSVEELSITVSH
NLLLEKKQRDKYVPKEKNALRFSTIEPDYVEKRAAQSSKLSVGYKITAGLEPKRKVLAALDCGGTQKPSPKVVR
AYIIYKLEPPENLFRSQEQGILVRLLLKKPDVYTDVLGLNCYLTQHKDTKRFMSVDGGNFGKCNDRFYKDEYSY
```

```
KPFVLVHMRKHIMLVVALEHDHALANRVRHVHHF
>sp|Q6GZX3|002L_FRG3G Uncharacterized protein 002L OS=Frog virus 3 (isolate
Goorha) OX=654924 GN=FV3-002L PE=4 SV=1
TPGGVYDVFADSSCKCVQTPCVPTEQRAQVGTKPGECGMKDKQQASSSAITDKDPQYKGGRSTTLPCTDSLGTT
LADMGRPPPAPDSRCRVAGGPVCWYMMYSWVSTLPEADVIVMGKKRPVIPTDEGDNVGGACPALAYVVTKPPAW
VQLAPGDGYSVRTRLKFCCVPIRPHASWNDYVPIAWKAVITMAAPGMRIFDCRPSCYDLDGEQLDIQKSNQDVV
ANINILDDDSYTPFFTTVPCPSPFCSAGDNPCHPTQKTKCPAANKTWAPHPNKCSPPSRLEPGAPDTAPKGVFT
NRTGTSQDYLLCSRYSTLFRRQFA
>sp|Q197F8|002R_IIV3 Uncharacterized protein 002R OS=Invertebrate
iridescent virus 3 OX=345201 GN=IIV3-002R PE=4 SV=1
TLHDASCPDSLIEGDRPIWMIGVEDGEIGKFRFLPWVCFCEATRSGDHRPNYGVLHICRLDTSFDKYHDPQEED
INWKFWYKPTSTERETIPDSLTISKSERPSKDMDIRFKTMTVWTELCDESLERYDANESDLLPEYEKDIVSLYS
VISLFVKFGEESAEQQVTPSSAYFDEFELDLLNNSWVLKGIQADDELFVHFMVFRIDIFDELDAYSLTDYLYLP
ELKWSAHLENDTNHYVMEPFVTSTQNEYMRSDYNEYDWTLNICFRRSENWQEEDPTDQKKLNDLEYVQSFDDKD
AELWEHIEEAVPTKPNMQLRIIPNRRAGGPIPDEGHVVDTSEMDSSFGPFATPSSSKSSDDNRSWEKRSRTFQI
QNDYLTWLLEQTIQWDVGRKRWSYTGYREGQNYDVTYWTPIDETMLTFMISHDPEPVDIAQPKMVNAFLNYDDK
ESENQKEDNNGAPL
...
```

De forma homóloga, para generar el dataset de control random, tenemos la función `random_to_file(in_file:str, out_file:str, seed = 1)-> None`, que sigue exactamente el mismo esquema que `shuffle_to_file`. Con semilla aleatoria = 1, el resultado de aplicarla a un archivo que contenga las secuencias anteriormente mostradas es el siguiente:

```
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R OS=Frog virus 3
(isolate Goorha) OX=654924 GN=FV3-001R PE=4 SV=1
FWDKESRSPHESAPQYARKIWEMAAAVAPHQATIRSVINIIRLAQVEGLEMTQTHLLWSTPWCSIPQGNVNDRT
EGTPNSASCLYWWPGGTIAHVVIPTNWNRKVYAPTFTVHQCSNWVHTQSNQNAVVYYMRYAIGVWGDVKCDDAR
AKIKEYGNLDGGKTGKLRMSSEALPMQHKEKTHYQAIAPFCGRTQVITRITAPWMQCLFHCLDDLLGQWKFAVC
WHWRGYTCPHNEHWQWHSEPLTSAMYPLAGHMWF
>sp|Q6GZX3|002L_FRG3G Uncharacterized protein 002L OS=Frog virus 3 (isolate
Goorha) OX=654924 GN=FV3-002L PE=4 SV=1
MQHKEPVNVSVIDCDFGGVHKMYTKNMMELIYSFWVEMCQDPFFMEYWPDWVIWDKNLWVERKECLAYADQECH
IWQGERGIGEQPVLVKSMEHMCAALYMRPMPDDMYREKHYVSNKGVHLHINDKDRDWMIPLCMGMWLIMEVYWY
DIIAIPDKVDDAALNSSFETMDTGGFFMLETYLFHFVCMYVHGLQVGCIKDRQVKVRVRAPMGKSAQWACNWFW
FFKKPWPGYDISAGTMTRIIMSSIQMVYKICDTNGTHLLLVNGRYDEYTWPGFKQHWCSPNPTGVCTDKEKDFY
DRIPQPGMRFYSHEQYVQELKIPV
>sp|Q197F8|002R_IIV3 Uncharacterized protein 002R OS=Invertebrate
iridescent virus 3 OX=345201 GN=IIV3-002R PE=4 SV=1
AHTRWAAYIKHGLFVHKLWKRGVNSQEHWPHLEAEWAVLFDTNWLQTNTMAERRNLVPMWSEPPHVAKYTHRYT
QLGRYTHNTAPWQPMYWDSILAQFPKGDYANKQVLFRKSGRTCKTEWQDNDRAGTGDPKYLHTHIMKDDTNRTV
CGLVKNYIPVPGSKYMIKYIAYPMQIKHDGWRWFYKRTGFFRNLPIEHLDEIPMSEGCCYAHCSTYRMKEYGEI
PISRPGIILRVWPHRKMSWEHDCAASMPWLHPGFAAPFVCWPKFDRLACVCTFCKEQDHASFKHRPMKKIICWW
GNQYVTCNVQVHVQDKYDKGEFCHQCCDTSTNEMCFVCRFPRATKDKMDLCPCKMFKPELEQITVHMMTPWSEF
RTVWTVALGHNPTMEQNFWDCLVMQLMNKMTTATEFMMMWDRKSRNPDWCFCTSWKIWMNNPLRYMVTGAFKIW
FEGQYCEVKEHKDW
...
```

Para generar estos dataset de control, de forma fácil y customizable, tomando cualquier dataset original; la herramienta provee la función `generate_working_files(dataset_name :str, exp:str, quantity:int) -> None` donde `dataset_name` es el nombre del dataset que se tomará como base para hacer el dataset de control; `exp` puede ser `"s"` ó `"r"`, e indica si se desea generar archivos *shuffled* ó *random* respectivamente y `quantity` indica cuántos archivos se desea realizar del tipo dado, el mínimo es 1 y el máximo es 99. Tener en cuenta que empezarán con la semilla aleatoria 1, luego con la 2, con la 3 y así hasta generar la cantidad de datasets deseada.

Si bien los dataset *shuffled* y *random* generados son los principales sobre los que haremos los experimentos porque nos permitirían responder la pregunta de esta tesis, es decir, si es posible diferenciar los casos *shuffled* y *random* contra el original usando alguna métrica de complejidad algorítmica. También se generaron datos de control adicionales para ver cómo se comportan las métricas en casos extremadamente sencillos; estos fueron el caso *single character*, donde se reemplaza la entrada por una secuencia de `"A"` del mismo largo, y el caso *sorted* que ordena en orden alfabético los caracteres de la entrada.

Para lo primero, tenemos `single_char_to_file(in_file:str, out_file:str) -> None` que toma un archivo `in_file` fasta, genera y guarda su variante *single character* en `out_file`. Si la primera de las entradas antes mencionadas se encuentra en `in_file`, en `out_file`, se encontrará lo siguiente:

```
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R OS=Frog virus 3
(isolate Goorha) OX=654924 GN=FV3-001R PE=4 SV=1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
```

Para lo segundo, de forma homóloga a la anterior, tenemos la función `sorted_to_file (in_file:str, out_file:str) -> None` que genera la versión *sorted* del archivo fasta `in_file` y lo guarda en `out_file`. Si tomamos la misma entrada que el caso anterior tomó, se obtiene:

```
>sp|Q6GZX4|001R_FRG3G Putative transcription factor 001R OS=Frog virus 3
(isolate Goorha) OX=654924 GN=FV3-001R PE=4 SV=1
AAAAAAAAAAAAACCCCDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEFFFFFFFGGGGGGGGGGGGGGGHHH
HHHHHHIIIIIIIIIIIIKKKKKKKKKKKKKKKKKKKKKKKKKKLLLLLLLLLLLLLLLLLLLLLLLLMM
MMMMNNNNNNNNPPPPPPPPPPPQQQQQQQQQRRRRRRRRRRRRRRSSSSSSSSSSSSTTTTTTTTVVVVVV
VVVVVVVVVVVVVVWWWWYYYYYYYYYYYYY
...
```

Recapitulando, en esta sección vimos como la herramienta nos permite adaptar un dataset según la necesidades del proyecto. Además, para dicho dataset, se cuenta con varias funciones para generar fácilmente datasets de control como *shuffled*, *random*, *sorted* y *single character*; estas últimas dos nos permiten entender mejor el funcionamiento de la métrica de complejidad utilizada, viendo como se comporta en casos triviales cuando las

secuencias analizadas son de un único carácter o una secuencia de caracteres ordenados, respectivamente. Cabe hacer foco en que, mientras que al caso *random* sólo se lo entiende como varias distribuciones aleatorias uniformes sobre el alfabeto de aminoácidos posibles, para los casos *shuffled* caben varias interpretaciones. Si bien la intención original al mezclar el orden de los aminoácidos es romper las estructuras que puedan existir con orígenes biológicos, funcionales y evolutivos que tienen las proteínas que se estudian; otra interpretación es que estas son secuencias *random* cuya distribución es la misma que la de las proteínas estudiadas, la cual está lejos de ser uniforme para todo el alfabeto.

Antes de presentar las métricas y los resultados, en la siguiente sección se explicará en profundidad los alcances y conceptos claves sobre la herramienta desarrollada y utilizada en este trabajo.

2.2. Más Sobre Cómo Funciona la Herramienta

Las secciones anteriores sirvieron de introducción para mostrar, en el caso de este trabajo, las capacidades de la herramienta. Esta está implementada de forma semejante al de un parser; sobre el archivo multifasta se van *parseando* cada una de las entradas de principio a fin. Para esto, se utiliza la biblioteca `Bio` de `Python`, que cuenta con la clase `SeqIO` que permite un manejo muy cómo de las entradas del archivo multifasta como objetos. Por ejemplo, `SeqIO.SeqRecord` nos permite leer el contenido de la entrada, mientras que con `SeqIO.description` podemos acceder al header. Aprovechando esto, el corazón de este sistema es la función `map_bio(in_file:`*`str`*`, function, auxVar:`*`Any`*`) -> `*`None`*, esta toma un dataset como archivo multifasta y lo *parsea* aplicando el map de la función que se le pasa; Lo interesante es la versatilidad de esta función, que admite funciones que tomen solo un `seq_record` como funciones que tomen un `seq_record` y otra variable auxiliar que pueda ser de utilidad. Es decir, toma funciones del tipo `f(seq_record:`*`SeqIO.SeqRecord`*`) -> `*`Any`* y también `f(seq_record:`*`SeqIO.SeqRecord`*`, aux:`*`Any`*`) -> `*`Any`*. Es el corazón de la herramienta porque es una función muy poderosa, versátil y es la forma que tienen de interactuar la mayoría de las funciones con el dataset; si no es que lo hacen manteniendo el mismo esquema de parsear el dataset.

Puntualmente, la herramienta consta de cuatro archivos `.py`, `fasta_utils.py` da varias funciones para manejar los datos en formato fasta: contar y mostrar las entradas, ordenarlas y filtrarlas (como se explicó anteriormente) además de las funciones que permiten generar las versiones *shuffled*, *random*, *single character* y *sorted* del dataset. Este archivo utiliza a su vez funciones misceláneas de más bajo nivel que se encuentran en `misc_utils.py`, como leer y escribir archivos manteniendo formato y consistencia, junto con la función `map_bio`, entre otras.

El archivo `complexity_metrics.py` provee las funciones de complejidad. Está gobernada por la clase `ComplexitySelector` que se construye con un indicador, y contiene información útil como nombre, prefijo y extensión para los archivos y la función de complejidad a la que hace referencia la clase. Tendrán extensión `.txt` aquellas funciones de complejidad que devuelvan una único valor y extensión `.csv` las que devuelvan una lista con más de un valor. Luego, se encuentran en este archivo las demás funciones de complejidad a las que se hace referencia a través del objeto antes mencionado. Si se desea agregar una nueva función de complejidad basta con agregar la función al archivo y asignarle un id en el objeto. Cabe remarcar que toda función de complejidad debe ser del estilo `f(seq:`*`str`*`) -> `*`Any`*, donde `seq` es un string que contiene la secuencia de aminoácidos de una entrada.

El archivo `main.py` es el archivo principal del proyecto que importa a todos los demás, con sus respectivas funciones. En esta se encuentran las funciones de más alto nivel que tienen que ver con la manipulación de varios sets de datos a la vez de forma eficiente. Dada una función de métrica de complejidad que toma la secuencia de aminoácidos como *str*, y devuelve algún valor o lista de valores como resultado, tenemos la función `complexity_to_list(in_file:str, complexity_id:str) -> list`, que toma un dataset y el id de una complejidad y calcula la complejidad de cada entrada del dataset y lo devuelve como lista. También tenemos `complexity_to_file_with_feedback(in_file:str, out_file:str, complexity_id:str) -> None` que de forma homóloga a la anterior aplica la función de complejidad a la que referencia el id a cada entrada de `in_file`, pero, que guarda en `out_file` los resultados a medida que se van computando. Y la función principal que es: `experiment(dataset_name:str, complexity_id:str, exp:str = "s_and_r", gen:bool = False, control:bool = True, quantity:int = 10, mode:str = "performance") -> None` donde `dataset_name` es el nombre del dataset sin extensión, este debe ser un archivo `.fasta` y encontrarse en la carpeta `data`; `exp` permite indicar si se desea trabajar sólo con datos *shuffled* (`"s"`) o *random* (`"r"`), por defecto trabaja con ambos; `gen` indica si se deben generar los dataset de control *shuffled*, *random*, *single character* y *sorted*, la primera corrida sobre un dataset nuevo estos deben generarse (`gen = True`), luego, no será necesario (`gen = False`), notemos que por defecto no los genera; `control` indica si se calcula la complejidad sobre los casos *single character* y *sorted*, por defecto sí los calcula; `quantity` va de 0 a 99 e indica la cantidad de archivos shuffled o random con los que se desea trabajar, por defecto es 10, que sea 0 indica que sólo se quiere trabajar sobre el archivo original; `mode` permite que el experimento se compute con dos modos distintos, `"performance"` computará más velozmente las métricas pero las cargará al archivo solamente al finalizar, mientras que `"feedback"` tardará más en computar la totalidad de la métrica de complejidad pero irá mostrando los resultados a medida que los va calculando, esta demora se debe a que debe abrir y cerrar un archivo cada vez que computa el resultado de una entrada, mientras que el modo *performance* sólo abre y cierra el archivo al final.

En cuanto a la eficiencia, este proyecto hace uso de la biblioteca `multiprocessing` de `Python`, que nos permite computar en paralelo tanto el cálculo de complejidad como la generación del dataset de control. Para evitar problemas de concurrencia, se computa un único archivo a la vez por núcleo (o thread si el procesador es multithreading). Como `Pyhon`, por defecto usa un único núcleo (o thread), esto nos permite aumentar la velocidad de cómputo enormemente. Esto funciona de la siguiente manera: se generarán cinco batches que se computarán sólo cuando termine de computarse el anterior, el primero sólo tendrá al archivo original, el segundo a todos los archivos *shuffled*, el tercero a todos los archivos *random*, el cuarto y el quinto tendrán los casos *single character* y *sorted* respectivamente si se desean calcular. El cómputo de los batches se repartirá en cuántos núcleos (o threads) haya disponibles. Cuando se quiere generar el dataset de control, los archivos *random* y los *shuffled* a generar se agruparán en dos batches, respectivamente. Primero se computará el batch *shuffled* y luego, cuando este termine, se computará el *random*. Esto se logra gracias a la función de la herramienta `multiprocess(function:list-> Any, data:list[list]) ->list`, donde `function` es la función que toma una lista de argumentos para realizar algún cálculo que se desee paralelizar, `data` es la lista de listas de argumentos que alimentará a la función recibida. La biblioteca `multiprocessing` nos permite saber cuántos núcleos (o threads) tiene disponible el equipo para hacer la asignación de recursos lo más

eficiente posible. Así, si por ejemplo, tenemos un procesador de un núcleo y dos threads, siendo `f(An, Bn, Cn) -> Dn` y ejecutamos `multiprocessing(f, [[A1, B1, C1], [A2, B2, C2], [A3, B3, C3]])`, esto generará batches en función del poder de cómputo disponible, entonces se ejecutará primero `f(A1,B1,C2)` en paralelo con `f(A2,B2,C2)` y, cuando estos terminen, se ejecutará `f(A3,B3,C3)`; `multiprocessing` devolverá una lista con los resultados, o sea, `[D1, D2, D3]`. Notar que `f(A3,B3,C3)` se comenzará a computar sólo cuando `f(A1,B1,C2)` y `f(A2,B2,C2)` se hayan terminado de computar en su totalidad.

Esta es la herramienta que se utilizará para computar las métricas de complejidad descritas en la siguiente sección, cuya documentación se encuentra detallada en profundidad en la sección 5. Anexo: Documentación.

En cuanto a los gráficos de este trabajo, las funciones que permiten elaborarlos se encuentran todas en este mismo proyecto, `misc_ploting.py` tiene algunas de las funciones principales útiles a la hora de graficar, `ploting_boxplot.ipynb` permite graficar distintos boxplots y `ploting.ipynb` permite generar todos los gráficos que se verán en la siguiente sección. Estos se encuentran comentados acordemente para fácil manipulación y generación de gráficos para las distintas métricas de complejidad. Adicionalmente, todo está diseñado para que agregar nuevas métricas sea lo más sencillo posible.

3. MÉTRICAS DE COMPLEJIDAD Y SUS RESULTADOS

En esta sección se explicarán cada una de las métricas de complejidad utilizadas y se verán los resultados de aplicarlas a cada uno de los casos de prueba anteriormente vistos, buscando que la métrica sea capaz en diferenciar el dataset original de los casos *shuffled* y *random*. Como el 95.3 % de los datos está por debajo de largo 900 inclusive y el 96.6 % está por debajo de largo 1000 inclusive, los gráficos fueron cortados en el largo 1000 y 900 según corresponda. Dado que los outliers se extienden hasta el largo 35213, esto permitió una visualización centrada en donde se encuentran la mayoría de los datos. Además, para mejorar la visualización, el eje de largo de la entrada se encuentra en escala logarítmica. Todos los siguientes gráficos fueron realizados con el código en `ploting.py` con funciones que pueden ser fácilmente configurables para elegir si mostrar o no los outliers, si el eje de largo es lineal o logarítmico; y para los histogramas, manipular el la cantidad de bloques que se ven en el gráfico.

Otra consideración es que se tienen 10 dataset *shuffled* y 10 dataset *random*, para poder mostrar esto de manera sencilla en gráficos de puntos, se muestran los promedios de cada entrada con sus homólogas y también se grafica el desvío estándar. Lo que sucedió en la práctica es que el desvío estándar que se vio era tan pequeño que no se percibe en los gráficos, con algunas excepciones que se discutirán más adelante.

3.1. Icalc

Esta es una métrica de complejidad desarrollada por Becher y Heiber[6] que presentan una métrica de complejidad en cadenas, llamada I-complejidad (Icalc) que es calculable en tiempo y espacio lineales. Esta mide la cantidad de subcadenas diferentes en una cadena dada, donde las cadenas menos complejas son aquellas compuestas por repeticiones de un único símbolo, mientras que las más complejas son las cadenas de De Bruijn. La implementación en Python de la complejidad descrita en el paper es la siguiente:

```python
def icalc(seq:str) -> float:
    b = []
    for i in range(len(seq)):
        mr = 0
        for j in range(i, mr, -1):
            if seq[j-mr:j] != seq[i-mr+1:i+1]:
                continue
            while i - mr >= 0 and seq[i-mr] == seq[j-mr-1]:
                mr += 1
        b.append(mr)

    res = 0
    for i in range(len(b)):
        res += 1.0 / (1.0 + b[i])
    return res / len(b)
```

Notemos que el resultado se encuentra normalizado, por lo que los resultados estarán

entre 1 (mayor complejidad) y 0 (menor complejidad), como es descrito en el paper, la complejidad final es de $O(n)$ siendo n el tamaño de la entrada.

Resultados

En la figura 3.1 podemos ver que Icalc, en términos generales, decrece en función del largo de la entrada. Luego del largo 100, esta métrica logra diferenciar entre los casos original, *single charcater* y *sorted*; estos últimos dos dando muy por debajo del original, como es de esperarse. A modo de techo sobre los casos original y *shuffled* se encuentran los resultados *random* considerando una muy baja dispersión y una superposición con los otros dos casos, esto indicaría que hay parte de los dataset original y *shuffled* que se ven completamente aleatorios desde el punto de vista del Icalc. Finalmente, para el caso *shuffled* podemos ver una clara superposición con el caso original, aunque este último tenga una dispersión mayor hacia valores de menor complejidad.

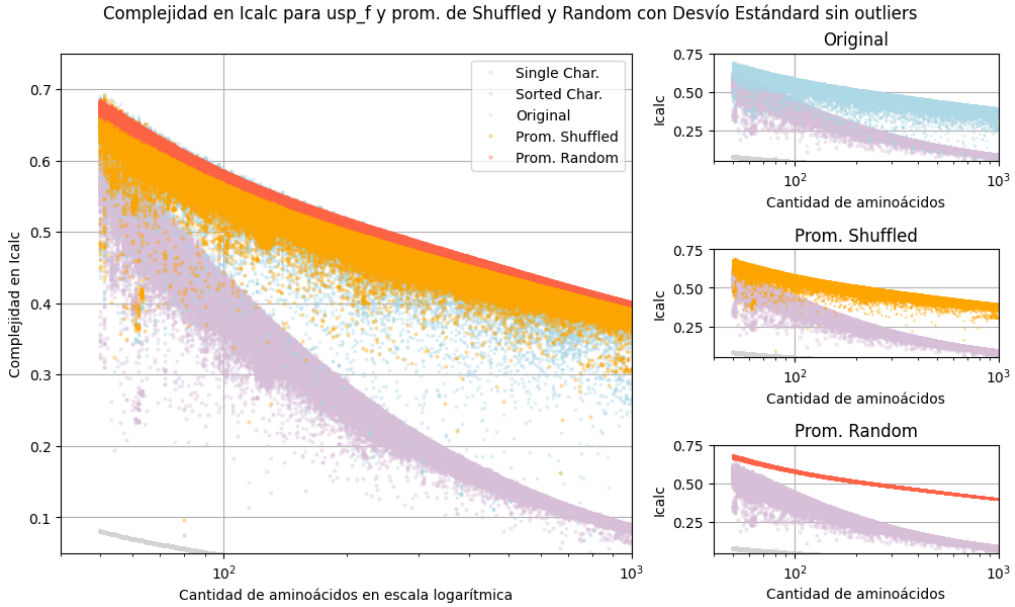

Fig. 3.1: Complejidad en Icalc en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

Para visualizar mejor si se logra diferenciar o no el caso original del *shuffled* o del *random*, en la figura 3.2 se divide el valor de cada resultado del dataset original por el promedio de sus homólogos *random* y *shuffled*; y se plotea el resultado en un histograma. Si se ve una concentración de puntos por fuera de la línea del 1, podemos concluir que hay una buena diferenciación; si la mayoría se encuentran sobre esta, no. Particularmente, para el caso *radom* la mayoría de los puntos se ubican levemente por debajo del 1 indicando que hay una leve diferenciación. Esto no ocurre con el caso *shuffled*, que de hecho están mucho más concentrados en la lína del 1. Consideremos que en ambos casos abundan los outliers que se encuentran alejados del 1 para todos los largos de entrada.

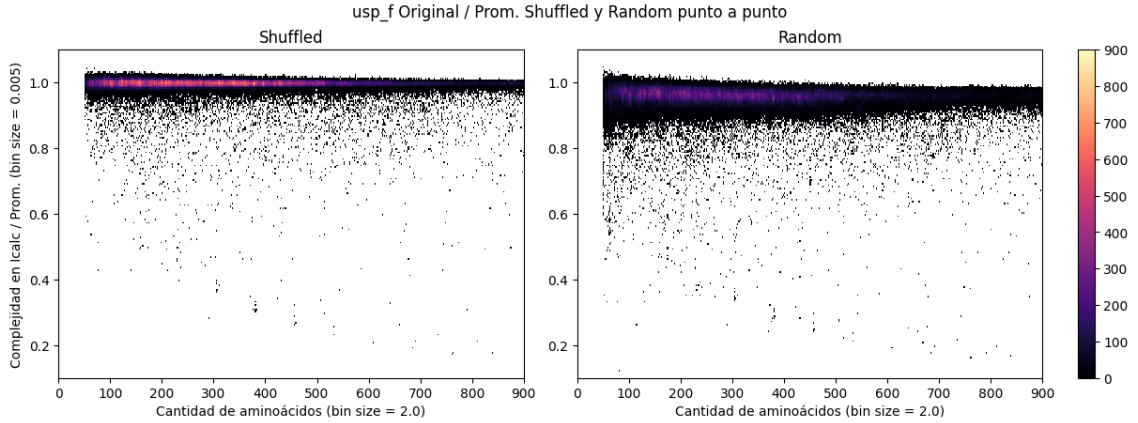

Fig. 3.2: Razón entre la complejidad en Icalc de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.005 por 2.

En líneas generales, Icalc permite notar una diferenciación del caso original con el caso *random* pero no así una con el caso *shuffled*. Teniendo en cuenta que existe una pequeña superposición entre los tres casos y existen abundantes outliers del caso original que puntuaron muy por debajo de las otras 2 variantes en esta métrica.

3.2. Discrepancia

Para este cálculo de complejidad se utiliza la siguiente definición tomada de la tesis de Ivo Pajor sobre Secuencias de De Bruijn con Discrepancia Mínima[8]. La discrepancia de una cadena es un número entero no negativo que indica la diferencia máxima, en cualquier subcadena, entre el número de apariciones del símbolo que más aparece y el símbolo que menos aparece en esa subcadena.

Formalmente: sea Σ un alfabeto y w una cadena sobre Σ. Denotemos $|w|_a$ al número de apariciones del símbolo a en w. Sea $S(w)$ el conjunto de todas las subcadenas de w. La discrepancia de una cadena w sobre el alfabeto Σ es una función $\Sigma - > \mathbb{Z}^*$ que se define como

$$discrepancia(w) = max_{s \in S}(max_{a \in \Sigma}|s|_a - min_{c \in \Sigma}|s|_c)$$

Por ejemplo, con $\Sigma = (0, 1)$, tenemos que:

- discrepancia($\underline{11111}0$) = 5
- discrepancia($\underline{1101}10$) = 3
- discrepancia($1\underline{00100}$) = 3

La lógica tras la implementación sigue el esquema del algoritmo de Kadane[1] que, dada una lista de enteros, permite encontrar la sublista que maximice la suma de los elementos en su contenido y devolver el resultado de esta suma en $O(n)$, consideremos que un algoritmo

[1] Explicación e implementación en (www.geeksforgeeks.org/largest-sum-contiguous-subarray/)

naive que haga lo propio tendría costo $O(n^2)$ al hacer la comparación todos con todos, por lo que este algoritmo es relevante. Entonces, con la lista w sobre el alfabeto Σ y dados dos elementos de Σ, a y c interpretamos a cada elemento de w como 1 si es igual a a, -1 si es igual a c y 0 si no. Entonces sobre la lista w se puede calcular $\max_{s \in S}(|s|_a - |s|_c)$, que implícitamente requiere que la sublista s tenga la mayor cantidad de apariciones de a y la menor cantidad de apariciones de c. Luego, si iteramos sobre todas las posibilidades de a y c del alfabeto Σ y devolvemos el máximo de los resultados, podemos tener completada la función de discrepancia(w).

La implementación en `Python` es la siguiente:

```python
def Kadane_for_2(seq:str, pos:str, neg:str) -> int:
    res = 0
    maxEnding = 0
    for i in range(len(seq)):
        to_add = 1 if seq[i] == pos else (-1 if seq[i] == neg else 0)
        maxEnding = max(maxEnding + to_add, to_add)
        res = max(res, maxEnding)
    return res

def discrepancy(seq:str) -> int:
    #alphabet = set("TGAC")                    #para ácidos nucleicos
    alphabet = set("ACDEFGHIKLMNPQRSTVWY")     #para aminoácidos
    res = 0
    for i in alphabet:
        remaining_alphabet = alphabet - {i}
        for j in remaining_alphabet:
            res = max(res, Kadane_for_2(seq, i, j))
    return res
```

En cuanto a la complejidad total, tenemos que $\Sigma = (A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y)$ y $|\Sigma| = 20 = k$, como se recorre la lista en $O(n)$ (con n largo de w) $k * k$ veces al ser necesario probar cada par de elementos pertenecientes a Σ, la complejidad final es $O(n * k^2) = O(n * 20^2) = O(400n) = O(n)$.

Resultados

En la figura 3.3 se ve que la complejidad crece de forma lineal para todos los casos (esto se ve curvado por la escala logarítmica). El caso *single character* hace la vez de techo para esta complejidad sin tener ningún tipo de superposición con los demás datasets, mientras que el caso *random* hace la vez de piso, logrando una completa separación con los demás datasets después del largo 500 aproximadamente. Luego, los casos original, *shuffled* y *sorted* se encuentran muy superpuestos, haciendo difícil diferenciarlos entre sí.

Anteriormente se menciona que los gráficos muestran el desvío estándar pero este no puede apreciarse por cuestiones coyunturales. A modo de ejemplo en la figura 3.4 se ve que en el caso *random* aumenta el desvío estándar en función del largo de las entradas (cantidad de aminoácidos); mientras que en el caso *shuffled* el desvío estándar es tan bajo que no es posible notarlo sin importar el largo de las entradas. En cuanto al desvío estándar, dicho caso *random* es un caso particular, para todas las demás métricas, tanto para los

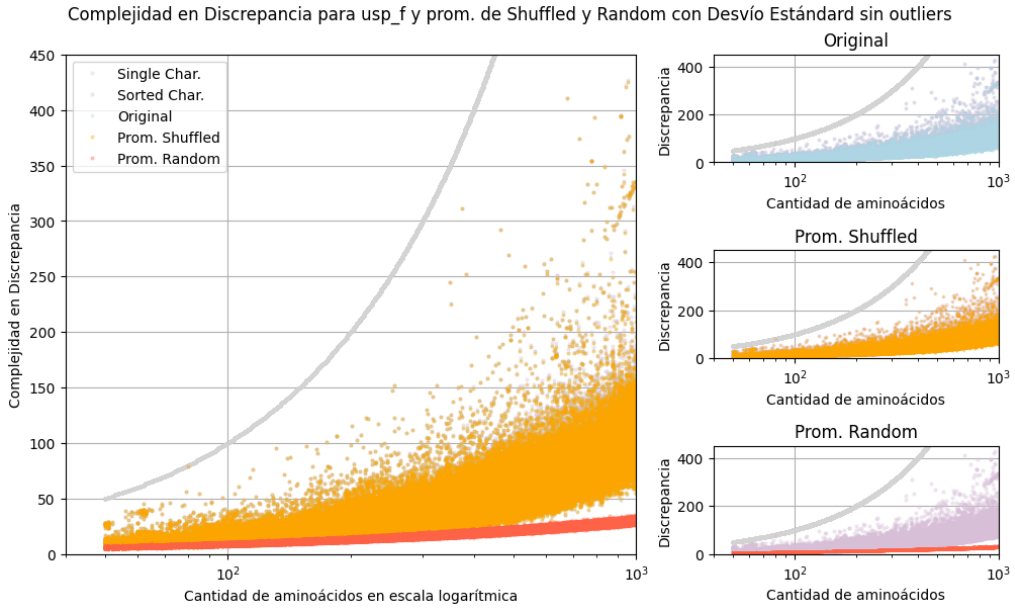

Fig. 3.3: Complejidad en Discrepancia en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

casos *shuffled* como *random* el desvío estándar es mínimo al punto no ser notable en las imágenes sin importar la cantidad de aminoácidos que se muestren.

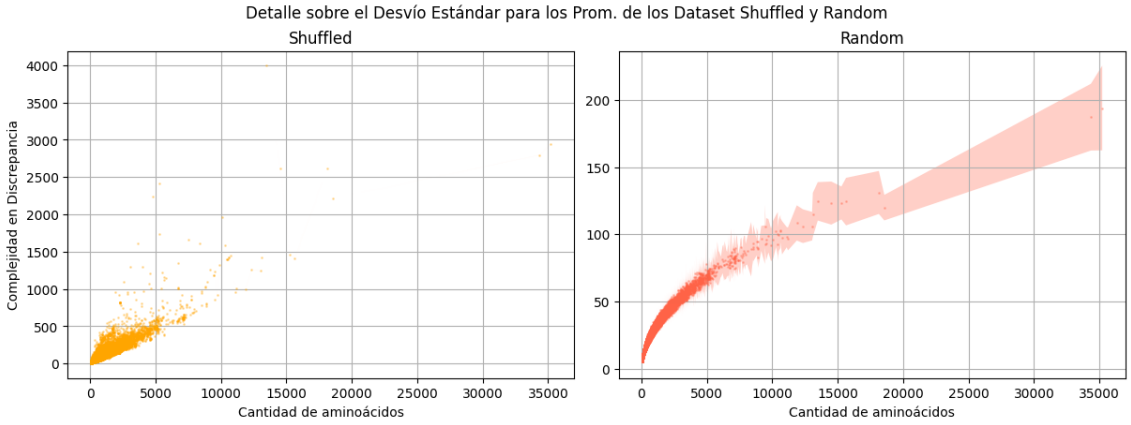

Fig. 3.4: Complejidad en Discrepancia en función del largo de la entrada, en escala lineal. Con desvío estándar mostrado como el sombreado del mismo color de los puntos. Para la totalidad del dataset `usp_f`, considerando outliers.

En la figura 3.5 podemos ver que en el caso *shuffled* hay una muy leve diferencia contra el caso original, esta se da al principio (menores largos) y se atenúa sobre el final (mayores largos) donde la diferenciación con el caso original es aún menor. Mientras que en el caso

random hay mucha dispersión en toda la muestra y este rápidamente logra diferenciarse del caso original, siendo que después del 300 ya no hay ningún punto en la línea del 1.

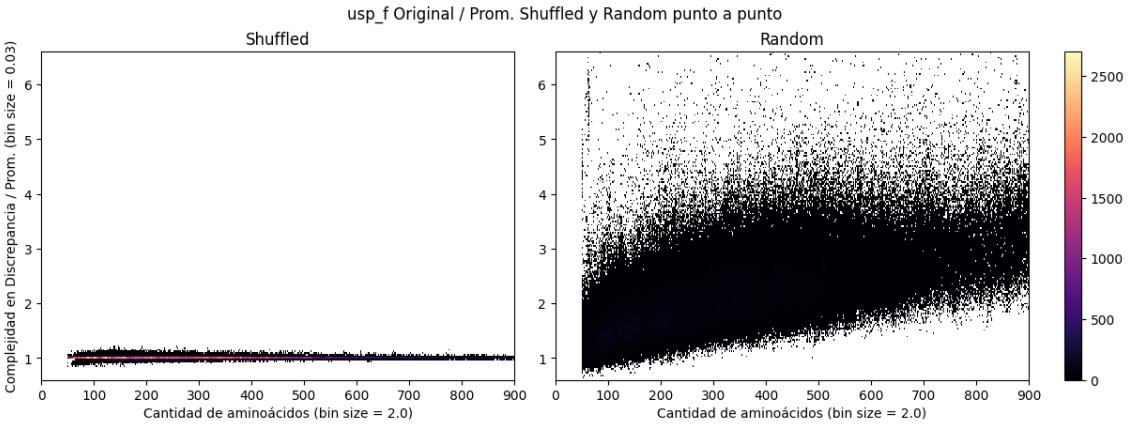

Fig. 3.5: Razón entre la complejidad en Discrepancia de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.03 por 2.

Concluyendo, Discrepancia logra una mejor diferenciación en el caso *random* que el Icalc, pero una mucha peor diferenciación en el caso *shuffled*. Con la interesante particularidad de que no es capaz de diferenciar el caso *sorted* del original tampoco.

3.3. Discrepancia en Bloque

Dado un largo de bloque b, la discrepancia en bloque calcula exactamente lo mismo que la discrepancia anterior pero tomando como símbolos las subsecuencias de largo b.

Formalmente: sea Σ un alfabeto, w una cadena sobre Σ y v_b una cadena de largo b sobre Σ. Denotamos $|w|_{v_b}$ al número de apariciones de la cadena v_b en w. Sea $S(w)$ el conjunto de todas las subcadenas de w. La discrepancia en bloque b de una cadena w sobre el alfabeto Σ es una función $\Sigma -> \mathbb{Z}^*$ que se define como

$$discrepancia\_en\_bloque(w, b) = max_{s \in S}(max_{v_b \in S}|s|_{v_b} - min_{u_b \in S}|s|_{u_b})$$

Notemos que el caso de la Discrepancia queda contenido en la Discrepancia en Bloques cuando el bloque es de tamaño 1.

En cuanto a la implementación, la idea es la misma que la de antes, usar el algoritmo de Kadane, un índice que recorre la secuencia una única vez, para mantenerlo en $O(n)$. Pero, veamos los siguientes ejemplos, buscando apariciones de AA en AAA, el resultado debe ser 1 y no 2, no hay que permitir el solapamiento de bloques; mientras que contando apariciones de AB - apariciones de BA en ABA el resultado debe ser 1 - 1 = 0 , es necesario el solapamiento. Entonces, vemos que con un único índice no es suficiente, precisamos de al menos dos. Por eso en el siguiente algoritmo se utilizan `pos_index` y `neg_index`, recordemos que cuando utilizamos la modificación del algoritmo de Kadane, sólo comparamos de a pares de símbolos en el alfabeto, considerando todas las combinaciones posibles. Cada índice se actualiza con la aparición de los símbolos que le corresponden a cada uno, los que restan

para `neg_index` y los que suman para `pos_index`. Esto implica que, a medida que el índice principal del algoritmo de Kadane avanza, contará la aparición de una secuencia positiva si dicho índice es mayor a `poss_index` y lo actualizará; lo mismo sucede para el caso negativo.

La implementación en `Python` es la siguiente:

```
def Kadane_for_2blocks(seq:str, pos:str, neg:str) -> int:
    res = 0
    maxEnding = 0
    pos_index = 0
    neg_index = 0
    for i in range(len(seq)-len(pos)+1):
        if seq[i:i+len(pos)] == pos and i >= pos_index:
            to_add = 1
            pos_index = i + len(pos)
        elif seq[i:i+len(pos)] == neg and i >= neg_index:
            to_add = -1
            neg_index = i + len(pos)
        else:
            to_add = 0
        maxEnding = max(maxEnding + to_add, to_add)
        res = max(res, maxEnding)
    return res

def discrepancy(seq:str, block_size:int = 1) -> int:
    #alphabet = set("TGAC")                    #para ácidos nucleicos
    alphabet = set("ACDEFGHIKLMNPQRSTVWY")    #para aminoácidos
    working_alph = alphabet.copy()
    res = 0
    for _ in range(block_size-1):
        working_alph = {i + j for i in working_alph for j in alphabet}
    for i in working_alph:
        remaining_alphabet = working_alph - {i}
        for j in remaining_alphabet:
            res = max(res, Kadane_for_2blocks(seq, i, j))
    return res
```

En cuanto a la complejidad computacional, si bien se mantiene que el algoritmo de Kadane recorre la secuencia una única vez, tenemos que ahora el alfabeto con el que se trabaja k depende del largo del bloque b, o sea, $k = 20^b$ lo que deja la complejidad total del algorimo en $O(n * 20^{2b})$, recordemos el algoritmo de Kadane tiene complejidad $O(n * 20^2)$, esto, como en el caso anterior reduce, teóricamente a $O(n)$. Pero en la práctica tenemos que para tamaño de bloque $b = 2$ se obtiene $20^4 = 160{,}000$ y si fuera $b = 3$ se obtiene $20^6 = 64{,}000{,}000$; por lo que podemos ver que si bien la complejidad es $O(n)$, el crecimiento es notorio. Esto se corresponde con lo que se ve en la práctica al querer aplicar este algoritmo a secuencias del dataset. Por ello, en este trabajo, no se hicieron experimentaciones con discrepancia en bloque 3 pero sí se experimentó con discrepancia en bloque 2 sobre una versión mucho más reducida del dataset original.

Resultados

Por esta razón para calcular esta complejidad altamente costosa, utilizamos una submuestra del dataset `usp_f`, `usp_987` que cosiste en todas las entradas de `usp_f` de largo 300 (del estudio de `usp_f` vimos que la media está alrededor de este número) más otras 2 secuencias de interés que se encuentran al principio de `usp_987`, con largo 393 y 317 respectivamente, estas son:

```
>sp|P04637|P53_HUMAN Cellular tumor antigen p53 OS=Homo sapiens OX=9606
GN=TP53 PE=1 SV=4
>sp|P25963|IKBA_HUMAN NF-kappa-B inhibitor alpha OS=Homo sapiens OX=9606
GN=NFKBIA PE=1 SV=1
```

El nombre `usp_987` hace referencia a que este dataset en total tiene 987 entradas, 985 de largo 300 y otras 2 mencionadas anteriormente.

Particularmente, `Cellular tumor antigen` da los resultados 9 para el original y, yendo del 1 al 10, tenemos para los casos *shuffled*: 7, 6, 6, 7, 7, 7, 6, 8, 6 y 6; y para los *random*: 5, 5, 5, 5, 7, 4, 5, 4, 4, y 5. Mientras que para `inhibitor alpha` tenemos 7 para el original, 6, 5, 10, 6, 7, 7, 6, 8, 9 y 7 para los *shuffled*; y 4, 4, 4, 4, 4, 4, 8, 5, 4 y 6 para los *random*.

En cuanto a los casos *Single Character* la primera entrada da 196, la segunda 158 y las demás de largo 300 dan 150. En cuanto a la versión *sorted* los resultados se encuentran todos por encima del bigote superior del resultado del caso original, por lo que se grafican a parte en la figura 3.6. Donde podemos ver que la media es de 18, el primer cuartil se encuentra en 16 mientras que el segundo se encuentra en 20 y los bigotes van de 12 a 26.

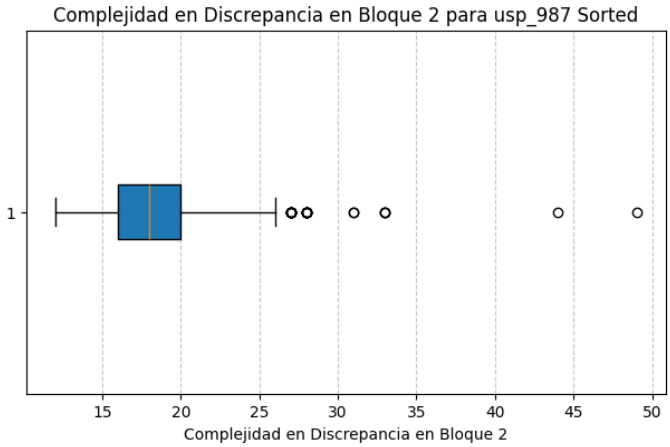

Fig. 3.6: Complejidad en Discrepancia en Bloque 2 para las secuencias del dataset *Sorted* de `usp_987`.

Al ser (casi) todas las entradas de tamaño 300, no tiene sentido graficar en función del largo de la entrada como hicimos anteriormente. Por ello, mostraremos boxplots de los resultados en la figura 3.7. En esta podemos ver que la media del dataset original es de 6, el primer cuartil está en 5, el tercer cuartil está en 8 y los bigotes están en 4 y 12. En el caso *random* podemos ver que hay un claro desacople con el original indicando una buena diferenciación con este. Mientras que el caso *shuffled* se comparten las medianas, primer cuartiles y (en la mayoría de los casos) bigote inferior con el caso original; si bien el caso original logra bigotes 20 % mayores que las variantes *shuffled* y el tercer cuartil se encuentra por encima de este el grueso de los datos se encuentra aún en cercana semejanza

con el caso original como para poder decir que son diferenciables.

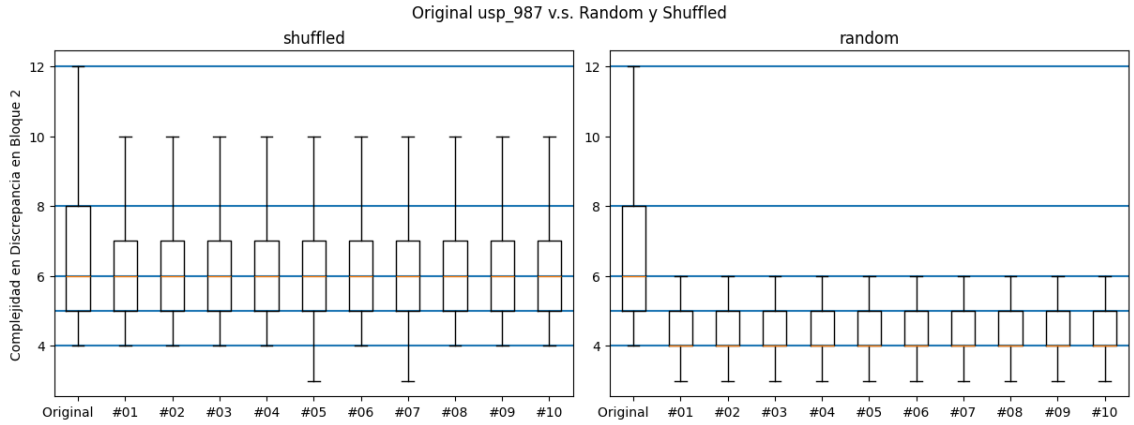

Fig. 3.7: Complejidad en Discrepancia en Bloque 2 para la secuencia original y sus 10 variantes *shuffled* (izquierda) y *random* (derecha). Las líneas azules indican la media, bigotes, primer y tercer cuartiles de la distribución original.

Cabe notar que la Discrepancia en Bloque 2 logra diferenciar el caso original del *sorted*, lo que no pudo lograrse con la Discrepancia *naive*. y si bien parece lograr una diferenciación levemente mejor que la Discrepancia *naive* en los casos *shuffled* y *random*, al ser una versión reducida del dataset la comparación no es válida.

3.4. Métricas provistas por OACC

Estas métricas son obtenidas gracias al trabajo de Soler, et.al, Gauvrit, et. al y Zenil, et. al [9, 10, 11] constan de una implementación en `R` de las métricas de *The Online Algorithmic Complexity Calculator* (OACC)[2], esta implementación permite calcular aproximaciones a través del *Block Decomposition Method* (BDM, basado en la Probabilidad Algorítmica[3]) a complejidades teóricas que no son computables como la complejidad algorítmica de Kolmogorov[3] o la profundidad lógica de Bennett[12], además también da como resultado la Entropía de Shannon, la entropía en segundo orden y el largo del archivo comprimido si se usara `gzip`. Cabe destacar que el BDM añade otras variables que deben ser consideradas y afectan el resultado tanto de Kolmogorov como Bennett, la primera y principal es el alfabeto. La OACC, si bien se adapta para funcionar en este proyecto, no cuenta con la posibilidad de un alfabeto de 20 símbolos. Entonces, lo que se utiliza es la conversión `"256 (utf-8)"` que consiste en tomar la secuencia en el estándar UTF-8 (8-bit Unicode Transformation Format, formato de codificación de caracteres Unicode e ISO 10646), pasarlo a binario y calcular sobre esto la complejidad con el BDM. Además se introducen los parámetros `Block Size` ($\mathbb{Z}$, que va de 2 a 12) y `Block Overlap` ($\mathbb{Z}$, que va de 0 a `Block Size` - 1). En el caso de este trabajo se utiliza `Block Size` $= 12$ y `Block Overlap` $= 0$, porque en la experimentación se vio que para una cadena dada es el caso de mayor velocidad para obtener resultado en comparación a las demás combinaciones posibles. Además, recordemos que estas son aproximaciones a complejidades teóricas no computables, por lo

[2] The Online Algorithmic Complexity Calculator (www.complexity-calculator.com)
[3] Algoritmic Probability (www.scholarpedia.org/article/Algorithmic_probability)

que es difícil ponderar supremacía de una combinación (de `Block Size`, `Block Overlap`) sobre otra y dicho estudio escapa al alcance de esta Tesis. A continuación se profundiza sobre cada una de estas métricas de complejidad provistas por OACC.

3.4.1. Kolmogorov - BDM algorithmic complexity estimation (bits)

La complejidad de Kolmogorov de un objeto, como un fragmento de texto, es la longitud del programa más corto que produce ese objeto como salida. Dicha longitud se mide en bits. Notemos que los *strings* aleatorios son incompresibles, por lo tanto, se considera que contienen mucha información. Sin embargo, la información aleatoria puede no ser muy útil desde un punto de vista computacional[13].

Resultados

En la figura 3.8 se ve que los casos original, *shuffled* y *random* crecen en función del largo de la entrada; se separan del caso *sorted* en 200 aproximadamente y que junto con *single character* se mantienen constantes en función del largo de la entrada, estando el primero entre 1000 y 2000 y el segundo muy por debajo de este. Podemos ver que al igual que sucedió en los casos anteriores, se pudo lograr cierta diferenciación entre el caso *random* y el original pero no pudieron diferenciarse el caso *shuffled* del original. Al igual que como sucedió con Icalc, parte de los dataset original y *shuffled* se ven como *random* para esta métrica.

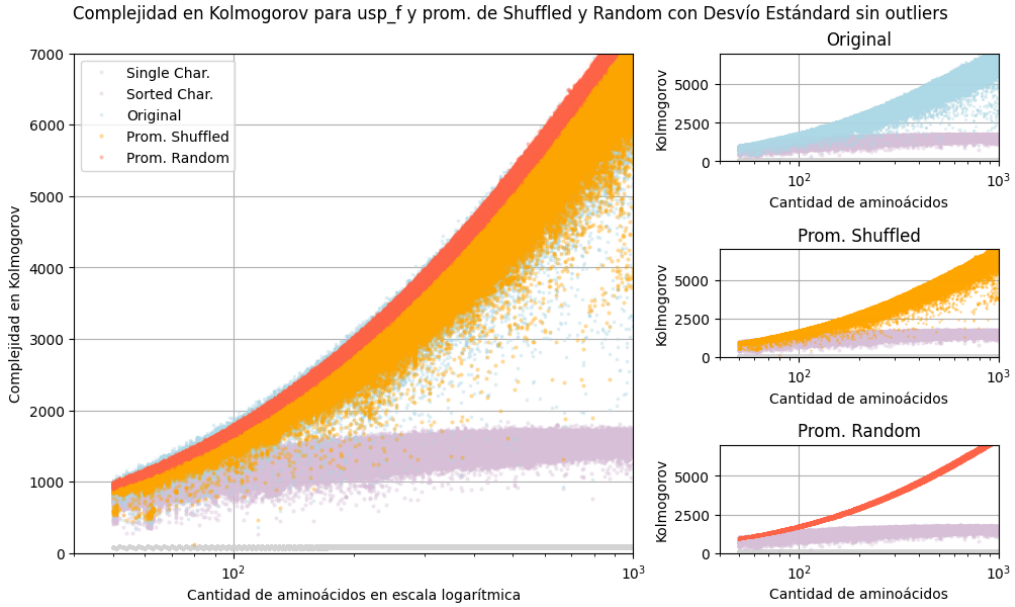

Fig. 3.8: Complejidad en Kolmogorov en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

Para confirmar lo dicho anteriormente, en la figura 3.9 podemos ver que no es posible

diferenciar el caso *shuffled* del original más allá del ruido al principio y que para el caso *random* se logra una insatisfactoria diferenciación a lo largo de todo los tamaños de entrada. Viendo que los puntos abandonan la línea del 1 luego del 600.

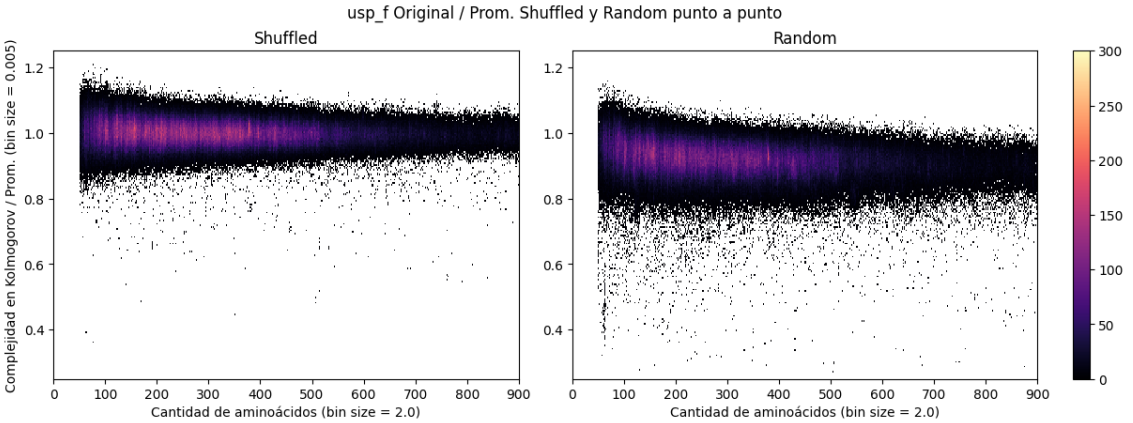

Fig. 3.9: Razón entre la complejidad en Kolmogorov de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.005 por 2.

Podemos concluir que el desempeño de esta métrica es similar a la del Icalc, sin lograr diferenciar con el original el caso *shuffled* y con una leve diferenciación para el caso *random*. De hecho se pueden ver similitudes mirando los histogramas 3.2 y 3.9. Salvando la diferencia de que Icalc presenta muchos más outliers en ambos casos y que Kolmogorov presenta una mayor dispersión al rededor de la línea del 1.

3.4.2. Bennnett - BDM logical depth estimation (steps)

La profundidad lógica es una medida de complejidad para *strings* individuales basada en la complejidad computacional de un algoritmo capaz de recrear una pieza de información dada. Se diferencia de la complejidad de Kolmogorov en que considera el tiempo de cómputo (medido en pasos) del algoritmo cuya longitud es cercana a la mínima, en lugar de enfocarse únicamente en la longitud del algoritmo mínimo.

Informalmente, la profundidad lógica de un *string* x con un nivel de significancia s es el tiempo requerido para calcular x mediante un programa que no sea más de s bits más largo que el programa más corto que genera x.

Formalmente, sea p^* el programa más corto que computa un *string* x en alguna computadora universal U. Entonces, la profundidad lógica de x con nivel de significancia s está dada por: $min\{T(p) : (|p| - |p^*| < s) \wedge (U(p) = x)\}$ donde $T(p)$ es el número de pasos de cómputo que el programa p realiza en U para producir x y *halt*.

Resultados

En la figura 3.10 se ve que todas los casos crecen en función del largo de la entrada. El caso *single charcater* es casi lineal, mientras que el crecimiento del caso original, *shuffled* y *random* se da de forma mucho más pronunciada que el caso *sorted*, el cual se separa de

estos alrededor del largo 300. Al igual que en Icalc y Kolmogorov, se tiene que parte de los casos original y *shuffled* se comportan como *random*, destacando que es posible ver outliers del caso original por encima del caso *random* y el *shuffled*, además de por debajo; esto no sucedió para ninguna de las métricas vistas anteriormente. Mientras que se logra cierta diferenciación entre el caso original y el *random*, esta no se logra con el caso *shuffled*.

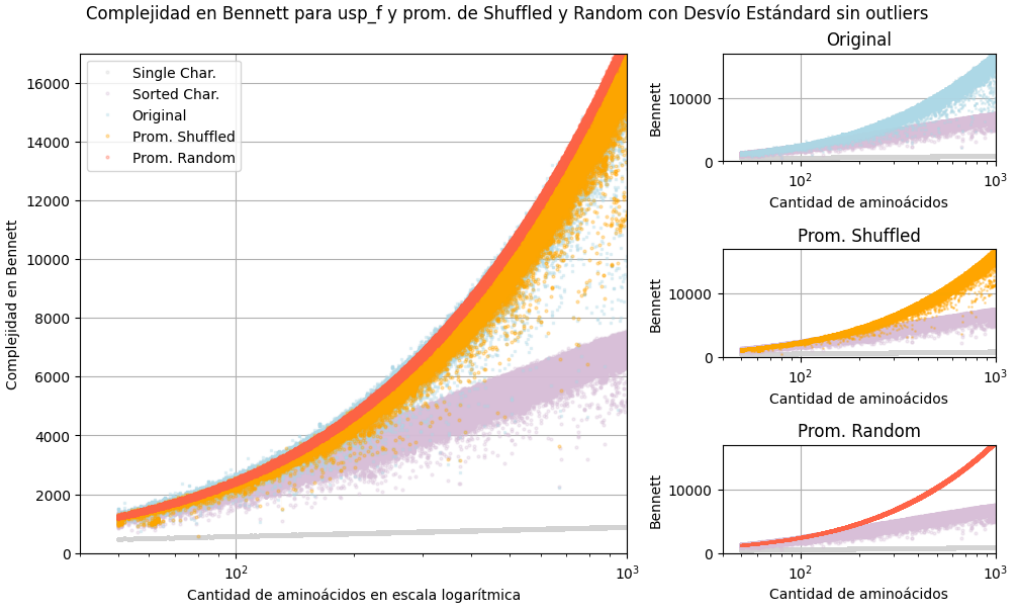

Fig. 3.10: Complejidad en Bennett en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

El histograma de la figura 3.11 confirma lo dicho anteriormente, comparando con Kolmogorov, tenemos una dispersión levemente menor tanto como los casos *shuffled* y *random* siendo que las figuras se ven muy similares. Puntualmente, en el caso *random* la mayoría de los puntos se encuentran por debajo del 1, por lo que hay una cierta diferenciación con el caso original.

Cabe destacar lo llamativo de la similitud de los resultados entre las métricas Kolmogorov y Bennett; siendo la segunda más sofisticada que la primera, se esperarían mejores resultados, lo que no sucedió en la práctica.

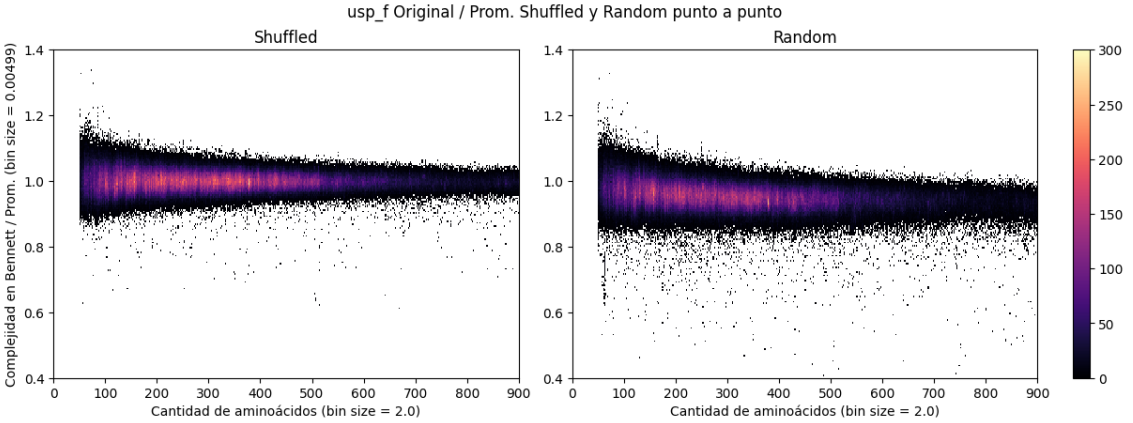

Fig. 3.11: Razón entre la complejidad en Bennett de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.005 por 2.

3.4.3. Shannon entropy (bit(s))

La entropía fue concebida originalmente por Shannon como una medida de la información transmitida a través de un canal de comunicación estocástico con alfabetos conocidos, y establece límites estrictos a las tasas máximas de compresión sin pérdida. Estos métodos forman la base de muchos, si no de la mayoría, de los algoritmos de compresión más comúnmente utilizados [11].

Entonces, es una medida de incertidumbre que mide la cantidad promedio de información considerando la probabilidad de cada símbolo. Formalmente, la entropía de un mensaje X, denotado por $H(X)$, es el valor medio ponderado de la cantidad de información de los diversos estados del mensaje: $H = -\sum_i p(x_i) log_2(x_i)$ siendo $p(x_i)$ la probabilidad del estado x_i para todos los i estados posibles.

Resultados

En la figura 3.12 se ve que la complejidad casi no aumenta después del largo 100. El caso *single character* hace la vez de piso siendo a penas divisible por estar tan pegado al cero. Mientras que el caso *random* hace la vez de techo separándose del caso original, *sorted* y el *shuffled*, que son idénticos, luego del largo 400 aproximadamente.

En el histograma de la figura 3.13 podemos confirmar lo que vimos anteriormente, el caso *shuffled* se ve como una perfecta línea sobre el 1, lo que confirma que cada resultado del caso *shuffled* es exactamente igual al resultado original. Con el caso *random* vemos que es distinto, pues después del 300 se logra una clara diferenciación del 1, si bien antes del 300 el centro de la distribución está claramente desviada hacia debajo del 1.

Notemos que es esperable que los casos original, *sorted* y *shuffled* resulten en valores idénticos para la Entropía de Shannon. Dado que esta mide la incertidumbre o aleatoriedad de una fuente de símbolos; entonces para el caso de una secuencia de aminoácidos, Shannon solo tiene en cuenta la distribución de frecuencias de los caracteres, no el orden en que aparecen.

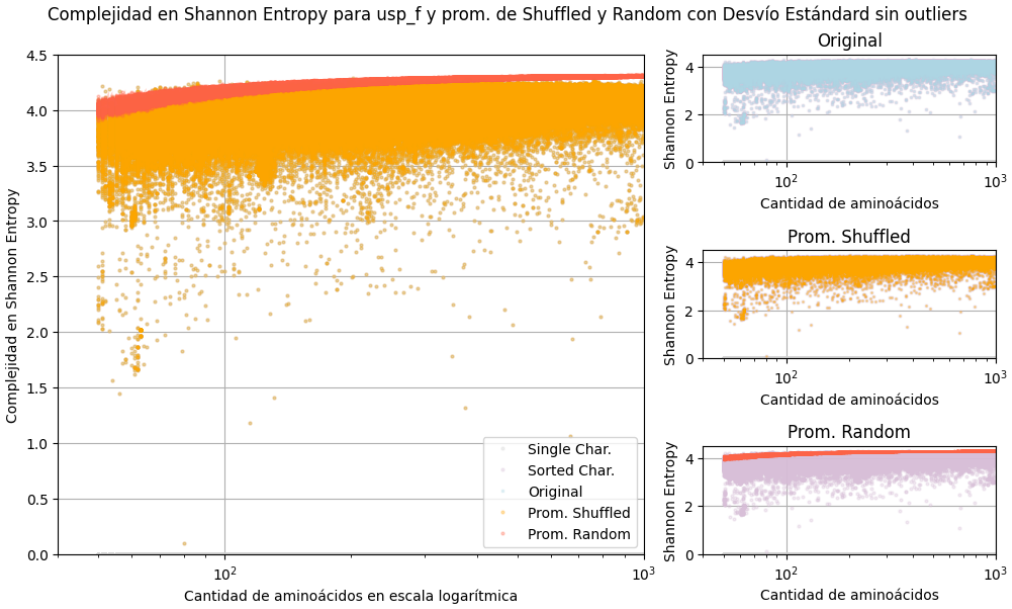

Fig. 3.12: Complejidad en Entropía de Shannon en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

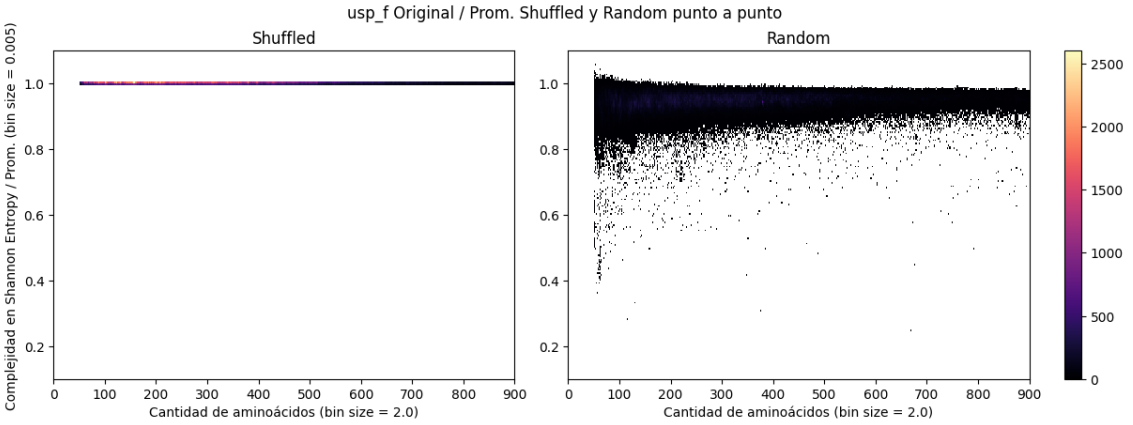

Fig. 3.13: Razón entre la complejidad en Entropía de Shannon de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.005 por 2.

3.4.4. Second order entropy (bit(s))

Al reemplazar la propiedad aditiva del valor medio de la entropía de Shannon por una propiedad exponencial del valor medio, Rényi[14] obtuvo una medida más general conocida como entropía de orden α: $H_\alpha \approx (1-\alpha)\log_2 \mathbb{E}_p\{p(x)^{\alpha-1}\}$

donde $p(x)$ es una función de densidad de probabilidad. Cuando $\alpha \to 1$, $H_\alpha \to H_1$ entonces $H_1 \approx - \mathbb{E}_p\{log_2\ p(x)\}$
i.e. H_1 es la entropía de Shannon. La entropía de segundo orden resulta cuando $\alpha = 2$, i.e. $H_2 \approx - log_2\ \mathbb{E}_p\{p(x)\}$.

La entropía de segundo orden permite considerar patrones más complejos, como la probabilidad de pares de símbolos consecutivos o bloques en lugar de símbolos individuales.

Resultados

En la figura 3.14 vemos que la complejidad en Entropía de Segundo Orden aumenta con el largo de la secuencia para los casos original, *shuffled* y *random*; mientras que el caso *single character* se mantiene constante pegado al cero y el caso *sorted* decrece con el largo de la secuencia. Notar que existe una buena diferenciación con este último y los demás y que a partir del largo 500 aproximadamente se diferencian el caso *random* con el *shuffled* y el original. Sin embargo, el caso *shuffled* y original siguen comportándose de forma similar, con la salvedad de que el original tiene una dispersión levemente mayor que el *shuffled*.

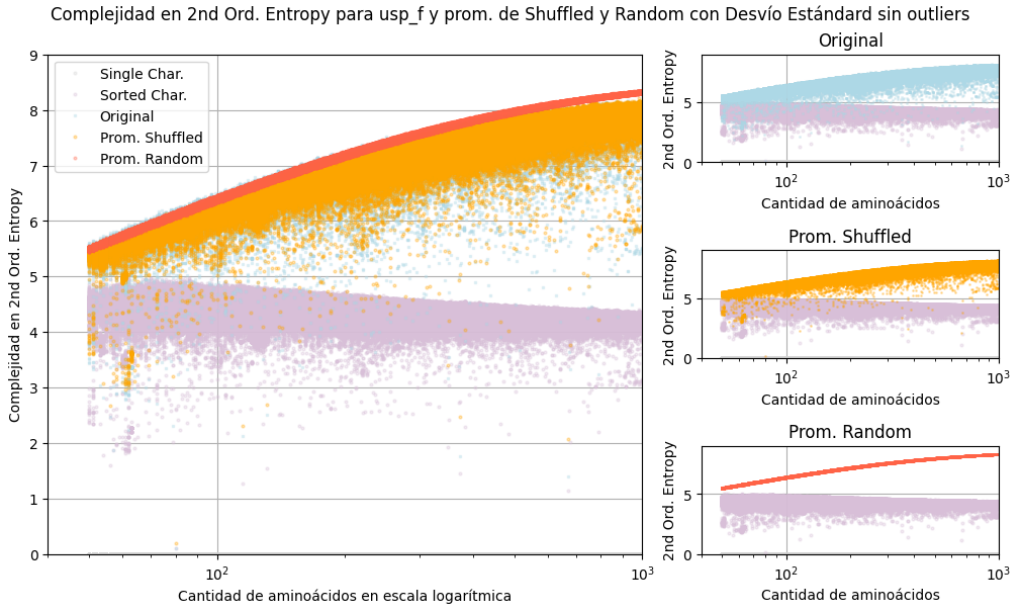

Fig. 3.14: Complejidad en Entropía de Segundo Orden en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

En el histograma de la figura 3.15 podemos ver que las razones entre el caso original y el caso *random* se encuentran por debajo del 1, quedando en su totalidad por debajo de este después del largo 300. En cuanto a la variante *shuffled* se ve que la mayoría de los puntos están en el 1, confirmando la falta de distinguibilidad que se veía anteriormente. Considerando que se encuentran menos outliers por debajo del 1, que los que se hubieran esperando sólo viendo la figura anterior (estos parecían ser comparables al caso del Icalc pero no lo son).

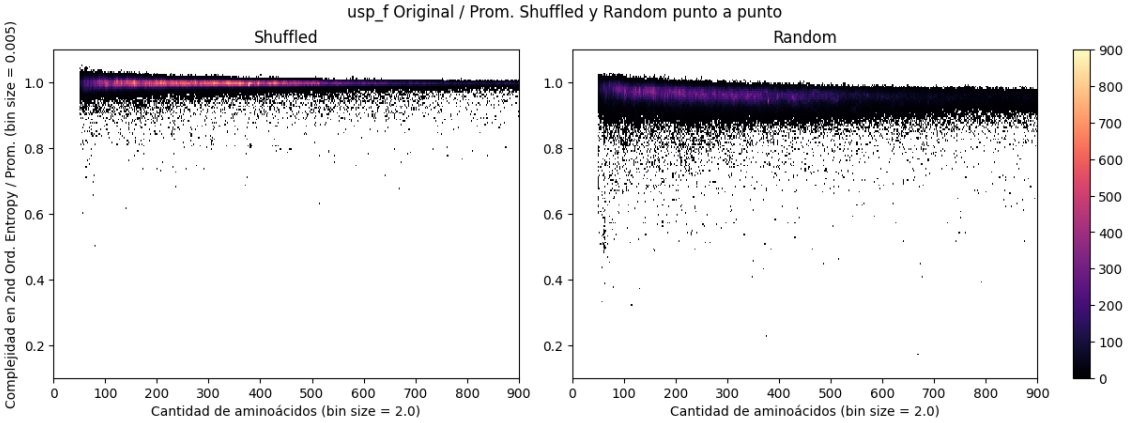

Fig. 3.15: Razón entre la complejidad en Entropía de Segundo Orden de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.005 por 2.

La Entropía de Segundo Orden después del umbral del largo 500 es capaz de diferenciar satisfactoriamente los casos originales de los *random*, como sucedió con métricas anteriores. Pero no es capaz de diferenciar el caso original del *shuffled*.

3.4.5. Compression length using gzip (bits)

Mide cuántos bits se necesitan para representar una secuencia después de ser comprimida por el algoritmo de `gzip`. Este algoritmo es ampliamente utilizado tanto en sistemas Unix como bases de datos comprimiendo archivos `http`, `CSS` y `JavaScript` generando un archivo `.gz` antes de enviarlos a un cliente; este programa hace buen uso de la sintaxis repetitiva de estos tipos de archivos. Es una métrica que permite reflejar la redundancia o aleatoriedad de las secuencias.

Resultados

En la figura 3.16 se puede ver cómo la complejidad en Largo de Compresión usando `gzip` tiene un crecimiento lineal en función del largo de las secuencias (que se ve curvado por la escala logarítmica) para los casos original, *shuffled* y *random*; mientras que se mantiene constante debajo de 500 para el caso *sorted* y a penas logra crecimiento en cuanto el caso *single character*. Este último se encuentra totalmente diferenciado de las demás métricas, mientras que el caso *sorted* se separa del original alrededor del largo 100. Se ve un solapamiento entre los casos casos original, *shuffled* y *random* hasta aproximadamente el largo 900; lo que indica que para esta métrica parte de los dataset original y *shuffled* se "ven" *random*. En cuanto al caso *shuffled* este se encuentra solapado con el original, con este último teniendo varios outliers por debajo del *shuffled*; sugiriendo de que el algoritmo de `gzip` logra encontrar estructuras subyacentes para reducir el tamaño de los casos originales, mucho mejor de lo que lo logra con los *shuffled*.

En el histograma de la figura 3.17 se ve que para el caso *random*, la mayoría de los puntos se encuentra levemente por debajo del 1 y totalmente por debajo de este, después

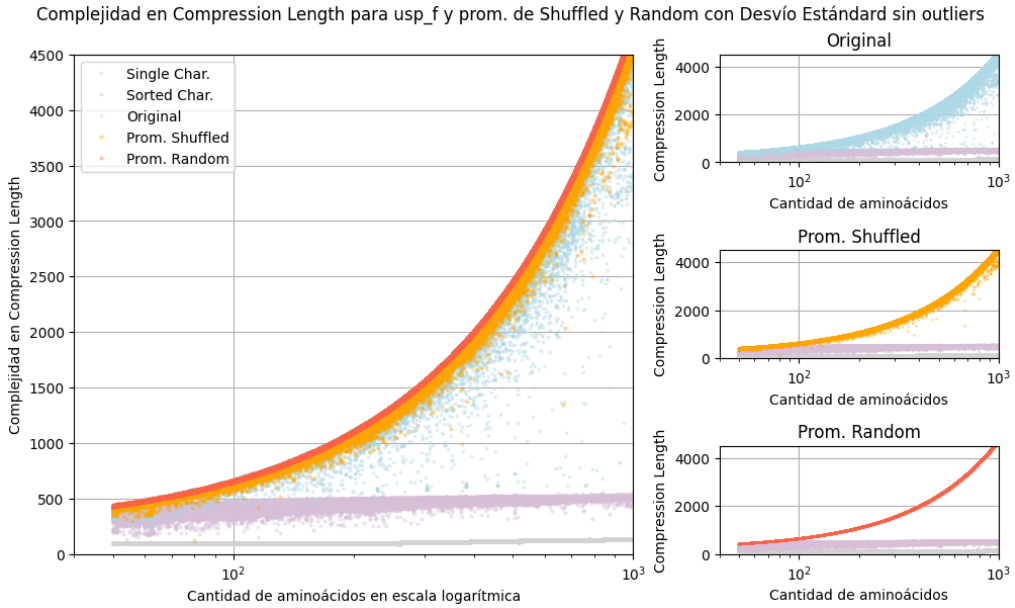

Fig. 3.16: Complejidad en Longitud de Compresión usando `gzip` en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los mandarina y tomate representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris. Las subfiguras a la derecha presentan una vista ampliada de los casos original, *shuffled* y *random*.

del largo 400. Mientras que en el caso *shuffled* la mayoría de los puntos se encuentran en la línea del 1, con una buena presencia de outliers por debajo de este, presentes en todos los largos de las secuencias.

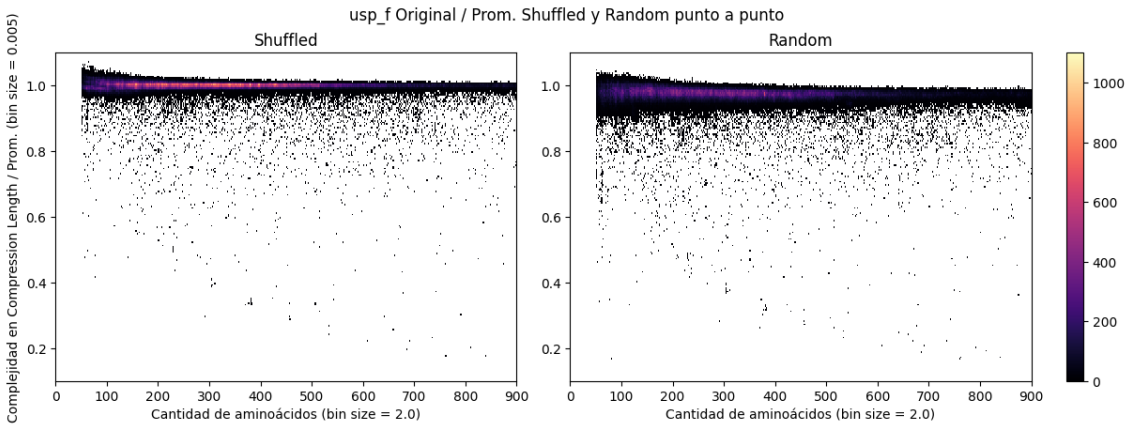

Fig. 3.17: Razón entre la complejidad en Longitud de Compresión usando `gzip` de las secuencias originales y el promedio de complejidad de los casos *shuffled* (izquierda) y *random* (derecha), en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 0.005 por 2.)

Los resultados de esta métrica son los esperables, pues *single character* es muy fácil

de comprimir y el tamaño del archivo resultante es mínimo; seguido por *sorted*. El caso *random* resulta tan difícil de comprimir como es posible. Y finalmente se ve que si bien para la mayoría de las secuencias originales no es posible distinguirlas de sus versiones *shuffled*; para muchas otras, el mezclado de aminoácidos logra destruir estructuras que son captadas por *gzip* y las utilizadas para generar archivos más pequeños, de ahí la abundancia de outliers en el histograma de la razón entre el caso original y el *shuffled*.

4. CONCLUSIONES Y TRABAJO FUTURO

En este trabajo pudimos filtrar, ordenar y estudiar secuencias de aminoácidos de proteínas de público acceso a través de la herramienta desarrollada. La cual nos permitió no sólo estudiar los casos originales, si no también generar casos adicionales de testeo pseudo-aleatorios de forma eficiente y fácilmente replicable, además de casos de control sencillos e intuitivos que ayudan a comprender las evaluaciones de las métricas. Esto se suma a un exhaustivo sistema de graficación que nos permite estudiar en profundidad los resultados obtenidos de las diferentes métricas. Se evaluaron con éxito 8 métricas distintas sobre dos datasets distintos y sus variantes *shuffled*, *random*, *sorted* y *single character*. Se mostró la facilidad con la que se cambia el dataset de trabajo y se agregan métricas adicionales. Demostrando esta herramienta ser un formidable aliado a la hora de estudiar la complejidad en secuencias de aminoácidos, sea cual sea el dataset y la métrica de complejidad que se quiera evaluar[7].

Vimos que para todas las métricas, los dataset *random* funcionaron como techo para las complejidades y *single character* como piso (exceptuando discrepancia que funcionó al revés). También, que después de cierto umbral, para muchas métricas *sorted* también funcionó como piso. Esto nos permite ver que las complejidades funcionan cómo esperamos y nos permite distinguir qué métricas son más acertadas para el estudio que queremos realizar. Es preferible una métrica que distinga a *sorted* como piso requiriendo el largo menor posible para diferenciarla, como Largo de Compresión que lo distingue luego del largo 100 o la Entropía de Segundo Orden que lo distingue desde el 50; que una que lo hace más tarde (como Bennett en 300) o que directamente no lo hace (como la Entropía de Shannon).

En cuanto a los resultados, vimos que varias métricas fueron capaces de diferenciar los casos *random* de los originales, entendiendo que tiene sentido que sean fáciles de diferenciar si se tiene en cuenta la distribución de aminoácidos es muy distinta entre estos datasets. Considerando que siempre hay superposición entre el caso original y el *random*, se concluye que siempre va a haber secuencias de aminoácidos de proteínas funcionales que se vean *random*, a menos para estas métricas. Aunque dicha superposición mejora en la mayoría de las métricas con el aumento del largo de las entradas; desapareciendo después del largo 500 en Discrepancia, 400 en Entropía de Shannon, 500 en Entropía de Segundo Orden y 900 en Largo de Compresión.

Sobre los casos *shuffled* no logramos separar satisfactoriamente a estos del caso original, a pesar de algunos outliers en algunas métricas particulares como Ical y Largo de Compresión. Esto nos lleva a la conclusión que al menos para las métricas elaboradas no existe diferencia entre secuencias de aminoácidos que generan proteínas funcionales y secuencias que no las generan siempre y cuando estas mantengan la misma distribución. Porque la interpretación alternativa de las secuencias *shuffled* es que son secuencias *random* sobre el alfabeto de los aminoácidos pero que mantienen la misma distribución que las proteínas funcionales conocidas a diferencia de las secuencias puramente *random* cuya distribución es uniforme.

4.1. Trabajo Futuro

- Nuevas Métricas. Uno de los objetivos de esta tesis es generar una herramienta que facilite el estudio de complejidad en secuencias de aminoácidos. Si bien no se logró encontrar *la métrica* que diferencie los casos *shuffled* de los originales, la herramienta generada puede utilizarse con nuevas métricas no probadas hasta ahora.

- Discrepancia en bloque.

  Computar la discrepancia en bloque 2 y 3 para la totalidad del dataset `usp_f`.

  Comprobar si es posible mejorar la complejidad de este algoritmo.

- Métricas para una distribución asimétrica. Vimos que muchas veces las métricas son capaces de diferenciar el caso *random* del *shuffled* y del original porque la distribución de estas últimas dos es distinta, dado que el *random* es uniforme sobre el alfabeto. Esto puede que sea un sesgo de las métricas, un posible trabajo futuro sería aplicar nuevas métricas o modificar las existentes que tengan en cuenta la distribución no uniforme de los aminoácidos en las proteínas.

- Estudio en profundidad del dataset. Durante la experimentación vimos que en la mayoría de los casos no es posible diferenciar entre el caso original y el *shuffled*, pero también vimos outliers de este suceso en diferentes métricas como Icalc y Longitud de Compresión, el estudio de estos outliers puede servir para entender el trasfondo de lo que está sucediendo y guiarnos mejor a métricas que puedan lograr el objetivo anterior.

- Mejora en el manejo del alfabeto. Como se explicó en la sección 3.4, The Online Algorithmic Complexity Calculator convierte el alfabeto de 20 símbolos a binario. Se puede adaptar esta herramienta para que trabaje de forma apropiada con un alfabeto de 20 símbolos y profundizar en los efectos de las diferentes combinaciones de `Block Size` y `Block Overlap`.

- Mejoras en eficiencia. Se puede mejorar la herramienta para que tanto en la generación de datasets de prueba como en la ejecución de experimentos estos se hagan de forma más eficiente. Hasta ahora la herramienta procesa primero el caso original, después los casos *shuffled*, luego los *random* y finalmente los dos de control. Teniendo que esperar que el anterior se compute en su totalidad para comenzar con el siguiente. Una mejora sería que se puedan procesar todos en una sola tanda haciendo mejor uso de la totalidad de los recursos del procesador.

5. ANEXO: DOCUMENTACIÓN DE LA HERRAMIENTA

5.1. Requerimientos y Versiones

Para la elaboración de este proyecto se utilizaron las últimas versiones de los programas y librerías disponibles a la fecha; estas son.

- `Python 3.9.6`
  - `Bio 1.8.0`
  - `rpy2 3.6.0`
  - `numpy 2.2.6`
  - `matplotlib 3.10.3`
- `R 4.5.0`
  - `acss 0.2-5`

5.2. Convenciones

- Los datos de trabajo, ya sean filtrados, shuffle, random, de control o de otro tipo se encuentran en la carpeta `data` con extensión `.fasta` junto con el archivo con los tamaños de las entradas del dataset, i.e. `"sizes_usp_f.txt"`.
- Los resultados se encontrarán en la carpeta `results` y el nombre del archivo estará conformado por: identificador de la complejidad medida de 4 letras `_` nombre del archivo del que se calcula la complejidad y `.txt` ó `.csv` dependiendo si el resultado de la métrica es uno o más parámetros respectimanete; i.e. `discr_usp_f_r06.txt` corresponde al resultado de aplicar la discrepancia al archivo `usp_f_r06.fasta`.
- `dataset_name`: *str*: nombre de dataset de trabajo, este debe encontrarse en la carpeta `data` y tener extensión `.fasta`, i.e. `"usp_f"`.
- `in_file`, `out_file`: *str*: dirección completa y extensión del archivo desde el lugar donde se ejecuta el archivo, i.e. `"data/usp_f.fasta"`.

5.3. Estructura del proyecto

- `data` – Carpeta que alberga todos los archivos de trabajo en formato `.fasta` y sus respectivas referencias en largos de secuencia con formato `.txt`, i.e. en `data` para el dataset `usp_f.fasta` se encuentra `sizes_usp_f.txt` con el largo de sus secuencias en el orden de aparición.
- `OACC-master` – Carpeta con parte del proyecto de The Online Algorithmic Complexity Calculator que permite calcular la complejidad de Kolmogorov, Bennett, Shannon, entropía de segundo orden y largo en compresión.
- `results` – Carpeta con los resultados de todas las experimentaciones.

- `complexity_metrics.py` – Archivo `Python` con las métricas de complejidad y la clase `ComplexitySelector` que permite seleccionar la complejidad con la que se va a trabajar junto con toda la información correspondiente a esa métrica.
- `fasta_utils.py` – Archivo `Python` que contiene herramientas para manejar los archivos fasta con mayo facilidad.
- `main.py` – Archivo `Python` con el desarrollo troncal del experimento.
- `misc_ploting.py` – Archivo `Python` con funciones auxiliares para la graficación de los resultados de los experimentos.
- `misc_utils.py` – Archivo `Python` con funciones auxiliares para implementaciones de `fasta_utils.py`.
- `polting_boxplot.ipynb` – Archivo `JupyterNotebooks` con boxplots principalmente relacionados a la distribución del largo del dataset.
- `polting.ipynb` – Archivo `JupyterNotebooks` con todos los gráficos predeterminados para cada complejidad.
- `workspace.py` – Archivo `Python` con ejemplos para experimentar sobre la herramienta.

5.4. Documentación de Funciones

En esta sección se presentarán todas las funciones de la herramienta[7] en orden alfabético.

Función: `complexity_from_files(dataset_name:`*str*`, complexity_id:`*str*`, quantity:`*int*` = 0, mode:`*str*` = "performance") -> None`

Descripción:
Aplica la complejidad marcada por `complexity_id` a cada entrada de los `quantity` archivos fasta referenciados por el `dataset_name`. El `mode` puede ser `"performance"` o `"feedback"` si se quiere que se computen más rápido o que se guarden en el archivo a medida que se van computando. `quantity = 0` indica el archivo original, esta es la opción por defecto.

Parámetros:

- `dataset_name` – Nombre del archivo fasta de referencia. Si este es `"usp_f"` se accederá al archvio `data\usp_f.fasta`.
- `complexity_id` – Identificador de complejidad, puede ser:

  `i` – Icalc.

  `d` – Discrepancia.

  `d2` a `d4` – Discrepacia en bloque 2 a 4.

  `b` – Block Decomposition Method de la Online Complexity Calculator, este devuelve las métricas Kolmogorov, Bennett, Shannon, entropía de segundo orden y largo en compresión a la vez.

- `quantity` – Entero positivo hasta 99 inclusive, si es 0 marca el archivo original.
- `mode` – Indica el modo de cómputo, pueden ser:

  `"performance"` – Computa todo y luego lo guarda, siendo más rápido.

  `"feedback"` – Guarda el resultado a medida que se va computando, llevará más tiempo el cómputo total de los datos.

Archivo: `main.py`

Ejemplo:

```
>>> complexity_from_files("usp_f", "i")
```

Calcula la compljidad Icalc del archivo `data\usp_f.fasta` y guarda el resultado en `results\icalc_usp_f.txt`.

```
>>> complexity_from_files("usp_f_s", "b", 3)
```

Calcula la compljidad Block Decomposition Method de los archivos `data\usp_f_s01.fasta`, `data\usp_f_s02.fasta` y `data\icalc_usp_f_s03.fasta` y guarda los resultados respectivos en los archivos `results\decom_usp_f_s01.csv`, `results\decom_usp_f_s02.csv` y `results\decom_usp_f_s03.csv`.

Función: `complexity_to_file_with_feedback(in_file:`*str*`, out_file:`*str*`, complexity_id) ->` *None*

Descripción:
Por cada entrada del archivo fasta `in_file` calcula su complejidad marcada por `complexity_id`, abre el archivo `out_file`, escribre el resultado y lo cierra; es decir, escribe los resultados uno a uno a medida que son computados, de ahí el "feedback". Esto se hace en orden.

Parámetros:

- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo donde se escribirán los resultados del cálculo de complejidad.
- `complexity_id` – Identificador de complejidad, puede ser:

  `i` – Icalc.

  `d` – Discrepancia.

  `d2` a `d4` – Discrepacia en bloque 2 a 4.

  `b` – Block Decomposition Method de la Online Complexity Calculator, este devuelve las métricas Kolmogorov, Bennett, Shannon, entropía de segundo orden y largo en compresión a la vez.

Archivo: `main.py`

Función: `complexity_to_list(in_file:str, complexity_id:str) -> list`

Descripción:
Aplica la complejidad marcada por `complexity_id` a cada entrada del archivo fasta `in_file`, en orden y guarda los resultados en una lista.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `complexity_id` – Identificador de complejidad, puede ser:

  `i` – Icalc.

  `d` – Discrepancia.

  `d2` a `d4` – Discrepacia en bloque 2 a 4.

  `b` – Block Decomposition Method de la Online Complexity Calculator, este devuelve las métricas Kolmogorov, Bennett, Shannon, entropía de segundo orden y largo en compresión a la vez.

Devuelve:
- `list` – Lista de los resultados del cálculo de la complejidad.

Archivo: `main.py`

Función: `copy_with_size(in_file:str, out_file:str, size:int) -> None`

Descripción:
Copia las entradas del archivo fasta `in_file` al archivo fasta `out_file` manteniendo el orden, pero reescribiéndolas con largo `size` (ver `write_with_size`).

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo fasta onde se reescribirán los datos.
- `size` – Largo con el que se escribirán las entradas del nuevo archivo fasta (ver `write_with_size`).

Archivo: `fasta_utils.py`

Función: `count_entries(in_file:str) -> int`

Descripción:
Cuenta la cantidad de entradas en del archivo fasta `in_file`.

Parámetros:
- `in_file` – Archivo fasta que se leerá.

Devuelve:
- `int` – Cantidad de entradas del archivo.

Archivo: `fasta_utils.py`

Ejemplo:

```
>>> count_entries("data/file_with_3_entries.fasta")
3
```

Función: `experiment(dataset_name, complexity_id:`*str*`, exp:`*str* `= "s_and_r", gen:`*bool* `= False, control:`*bool* `= True, quantity:`*int* `= 10, mode:`*str* `= "performance") -> None`

Descripción:
Función principal para realizar experimentos, dado un nombre de dataset `datast_name` y un identificador de complejidad `complexity_id`; permite calcular la complejidad indicando con `exp` si se quieren calcular los casos random, shuffled o ambos, el caso por default es ambos. También se puede indicar con `gen` si estos casos random, shuffled y de control desean ser generados o no, por defecto no se generan. `control` indica si se deben tener en cuenta los casos de control. El `mode` puede ser `"performance"` o `"feedback"` si se quiere que se computen más rápido o que se guarden en el archivo a medida que se van computando.

Parámetros:

- `dataset_name` – Nombre del archivo fasta de referencia. Si este es `"usp_f"` se accederá al archvio `data\usp_f.fasta`.
- `complexity_id` – Identificador de complejidad, puede ser:

  `i` – Icalc.

  `d` – Discrepancia.

  `d2` a `d4` – Discrepacia en bloque 2 a 4.

  `b` – Block Decomposition Method de la Online Complexity Calculator, este devuelve las métricas Kolmogorov, Bennett, Shannon, entropía de segundo orden y largo en compresión a la vez.
- `exp` – Indica si se generan (depende de `gen`) y se computan los casos random o shuffle.

  `"s_and_r"` – Ambos shuffle y random.

  `"s"` – Sólo casos shuffle.

  `"r"` – Sólo casos random.
- `gen` – Indica si se generan o no los casos de control.

  `True` – Se generan los shuffle si `exp` es `"s"` o `"s_and_r"` y se generan los random si `exp` es `"r"` o `"s_and_r"`.

  `False` – No se genera ninguno.
- `control` – Indica si se calcula la complejidad de los casos de control.

  `True` – Se calculan.

  `False` – No se calculan.
- `quantity` – Entero positivo hasta 99 inclusive, si es 0 marca el archivo original.
- `mode` – Indica el modo de cómputo, pueden ser:

  `"performance"` – Computa todo y luego lo guarda, siendo más rápido.

  `"feedback"` – Guarda el resultado a medida que se va computando, llevará más tiempo el cómputo total de los datos.

Archivo: `main.py`

Función: `filter_to_file(in_file:str, out_file:str, criteria) -> None`

Descripción:
Copia cada entrada del archivo fasta `in_file` y al arcivo fasta `out_file` siempre y cuando haga verdadera la función `criteria`. Esta es del tipo `criteria(seq_record:SeqIO.SeqRecord) -> bool`.

Parámetros:

- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo fasta donde se escribirá la secuencia.
- `criteria` – Función del tipo `criteria(seq_record:SeqIO.SeqRecord) -> bool`.

Archivo: `fasta_utils.py`

Ejemplo de uso:

```
>>> filter_to_file("data/uniprot_sprot.fasta", "data/usp_f.fasta",
    lambda seq_record : len(seq_record) >= 50)
```

Función: `generate_control_files(dataset_name:str) -> None`

Descripción:
Dado el nombre un dataset `dataset_name` genera dos archvivos de control usando `single_char_to_file` y `sorted_to_file`. El primero copia el archivo original reemplazando todos los caracteres de las entradas por `"A"` y el segundo copia el archivo original reemplazando las entradas por una versión ordenada alfabéticamente de las mismas.

Parámetros:

- `dataset_name` – Nombre del archivo fasta de referencia. Si este es `"usp_f"` se accederá al archvio `data\usp_f.fasta`.

Archivo: `main.py`

Ejemplo de uso:

```
>>> generate_control_files("usp_987")
```

Generará los archivos `data\singl_usp_987.fasta` y `data\sorte_usp_987.fasta`, con contenido como se describió anteriormente.

Función: `generate_working_files(dataset_name:str, exp:str, quantity :int) -> None`

Descripción:
Dado el nombre un dataset `dataset_name` genera `quantity` archivos de trabajo, estos pueden ser shuffled o random dependiendo si `exp` es `"s"` ó `"r"` respectivamente. Esto utiliza `multiproces` por lo que es más eficiente. Las semillas de generación aleatoria serán de 1 a `quantity` inclusive.

Parámetros:

- `dataset_name` – Nombre del archivo fasta de referencia. Si este es `"usp_f"` se accederá al archvio `data\usp_f.fasta`.
- `exp` – Indica qué archivos se generan, puede ser:

  `"s"` – Shuffled

  `"r"` – Random
- `quantity` – Natural entre 1 y 99 inclusive.

Archivo: `main.py`

Ejemplo:

```
>>> generate_working_files("usp_f", "r", 5)
```

Tomando de base `data\usp_f.fasta`, calculará 5 datasets de trabajo random, con semilla aleatoria del 1 al 5, generando los siguientes archivos:

```
data\usp_f_r01.fasta
data\usp_f_r02.fasta
data\usp_f_r03.fasta
data\usp_f_r04.fasta
data\usp_f_r05.fasta
```

Función: `make_name(prefix:`*str*`, num:`*int*`, sufix:`*str*` = "") -> `*str*

Descripción:
Concatena `prefix` con el número `num` (poniendo un `"0"` adelante si es menor a 10) y con el `sufix`.

Parámetros:

- `prefix` – Prefijo.
- `num` – Número entero del 0 al 99.
- `sufix` – Sufijo.

Devuelve:

- `str` – Concatenación de los parámetros recibidos.

Archivo: `misc_utils.py`

Ejemplo:

```
>>> make_name("usp_f_s", 2, ".fasta")
usp_f_s02.fasta
```

Función: `map_bio(in_file:str, function, aux_var = False) -> None`

Descripción:
Dado el archivo fasta `in_file`, permite aplicarle `function` a cada entrada del archivo, con la posibilidad de tener en cuenta una variable auxiliar. La función `function` es del tipo `function(seq_record:SeqIO.SeqRecord) -> Any` ó `function(seq_record:SeqIO.SeqRecord, aux_var:Any) -> Any`.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `function` – Función que se aplicará a cada entrada del archivo. Puede ser del tipo:

  ```
  function(seq_record:SeqIO.SeqRecord) -> Any
  function(seq_record:SeqIO.SeqRecord, aux_var:Any) -> Any
  ```

- `aux_var` – Variable auxiliar que puede llegar a precisar la función.

Archivo: `misc_utils.py`

Función: `multiprocess(function, data:list) -> list`

Descripción:
Aplica la función `function` a cada elemento de la lista `data`. Esto se hace seccionando los datos por cuantas cores o threads tengan los procesadroes del equipo y generando un proceso hijo por cada uno de estos logrando un cómputo óptimo. En la sección 2.2 se da una muy buena explicación de esto. En el proyecto esto se usa con funciones auxiliares que toman una lista de datos relevantes y aplica la función de complejidad (denotada por uno de los datos de la lista) según corresponda.

Parámetros:
- `function` – Función del tipo `function(data:any) -> any`.
- `data` – Lista con los datos a los que se le aplicará la función.

Devuelve:
- `list` – Lista con los resultados de la función.

Archivo: `main.py`

Función: `random_to_file(in_file:str, out_file:str, seed = 1) -> None`

Descripción:
Por cada entrada del archivo fasta `in_file`, manteniendo el orden, copia el header y tomando `seed` genera una distribución aleatoria uniforme del alfabeto de aminoácidos manteniendo el largo de la entrada original.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo donde se escribirá la secuencia.
- `seed` – Semilla de aleatoriedad.

Archivo: `misc_utils.py`

Función: `read_list_from_file(in_file:str) -> list`

Descripción:
Lee el archivo `in_file` que puede ser `.txt` si tiene una entrada por línea ó `.csv` si tiene más de una entrada por línea (separada por comas). Si es un `.txt` lo devuelve como una lista y si es un `.csv` lo devuelve como una lista de listas.

Parámetros:
- `in_file` – Archivo que se leerá.

Devuelve:
- `list` – Lista ó lista de listas con los datos leídos.

Archivo: `misc_utils.py`

Función: `save_list_to_file(l:list, out_file:str) -> None`

Descripción:
Abre el archivo `out_file` como Write, escribe `l` en este y lo cierra.

Parámetros:
- `l` – Contenido a escribir en el archivo.
- `out_file` – Archivo donde se escribirá la lista.

Archivo: `misc_utils.py`

Función: `show_entry(seq_record:SeqIO.SeqRecord) -> None`

Descripción:
Dado una entrada de un archivo fasta `seq_record`, la muestra por consola con su header, contenido y largo de la entrada.

Parámetros:
- `seq_record` – Entrada de un archivo fasta.

Archivo: `fasta_utils.py`

Función: `shuffle_to_file(in_file:str, out_file:str, seed = 1) -> None`

Descripción:
Por cada entrada del archivo fasta `in_file`, manteniendo el orden, copia el header y tomando `seed` mezcla los aminoácidos de la entrada. Si no se provee un valor de semilla este será 1.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo donde se escribirá la secuencia.
- `seed` – Semilla de aleatoriedad.

Archivo: `fasta_utils.py`

Función: `single_char_to_file(in_file:str, out_file:str) -> None`

Descripción:
Por cada entrada del archivo fasta `in_file`, manteniendo el orden, copia el header y escribe una secuencia de "A" de largo de la entrada original. Esto sirve como dataset de control.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo donde se escribirá la secuencia.

Archivo: `fasta_utils.py`

Función: `size_to_list(in_file:str) -> list`

Descripción:
Guarda el largo de cada entrada del archivo fasta `in_file` en una lista.

Parámetros:
- `in_file` – Archivo fasta que se leerá.

Devuelve:
- `list` – Lista de `int` con el largo de cada entrada.

Archivo: `fasta_utils.py`

Función: `sort_to_file(in_file:str, out_file:str) -> None`

Descripción:
Copia cada entrada del archivo fasta `in_file` y al archivo fasta `out_file` ordenada de menor a mayor en largo de entrada.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo fasta donde se escribirá la secuencia.

Archivo: `fasta_utils.py`

Función: `sorted_to_file(in_file:str, out_file:str) -> None`

Descripción:
Por cada entrada del archivo fasta `in_file`, manteniendo el orden, copia el header y escribe la secuencia ordenando los caracteres de la secuencia en orden alfabético. Esto sirve como datos de control.

Parámetros:
- `in_file` – Archivo fasta que se leerá.
- `out_file` – Archivo donde se escribirá la secuencia.

Archivo: `misc_utils.py`

Función: `write_with_size(seq:`*`str`*`, out_file:`*`str`*`, size:`*`int`*` = 100) -> `*`None`*

Descripción:
Escribe en el archivo `out_file` la secuencia `str` agregando un `\n` cada `size` caracteres. El archivo debe estar abierto de antemano. El valor por defecto es un salto de línea cada 100 caracteres.

Parámetros:

- `seq` – Secuencia a escribir en el archivo.
- `out_file` – Archivo donde se escribirá la secuencia.
- `size` – Entero positivo que indica cada cuánto se hará un salto de línea.

Archivo: `misc_utils.py`

Bibliografía

[1] Olaf Weiss, Miguel A Jiménez-Montaño, and Hanspeter Herzel. Information content of protein sequences. *Journal of theoretical biology*, 206(3):379–386, 2000.

[2] Pablo Turjanski and Diego U. Ferreiro. On the natural structure of amino acid patterns in families of protein sequences. *The Journal of Physical Chemistry B*, 122(49):11295–11301, 2018. PMID: 30239207.

[3] Andrei N Kolmogorov. Three approaches to the quantitative definition ofinformation'. *Problems of information transmission*, 1(1):1–7, 1965.

[4] Gregory J Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM (JACM)*, 22(3):329–340, 1975.

[5] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

[6] Verónica Becher and Pablo Ariel Heiber. A linearly computable measure of string complexity. *Theoretical Computer Science*, 438:62–73, 2012.

[7] Mariano Oca. Estudio de complejidad en secuencias de aminoácidos de proteínas. https://github.com/marianoOca/tesis, 2025. Tesis de Lienciatura en Cs. de la Computación.

[8] Ivo Pajor. Secuencias de de bruijn con discrepancia mínima. Tesis de lic. en cs. de la computación, Universidad de Buenos Aires, 2024. Available at https://gestion.dc.uba.ar/media/academic/grade/thesis/tesis_RgHuGM3.pdf.

[9] Fernando Soler-Toscano, Hector Zenil, Jean-Paul Delahaye, and Nicolas Gauvrit. Calculating Kolmogorov complexity from the output frequency distributions of small Turing machines. *PloS one*, 9(5):e96223, 2014.

[10] Nicolas Gauvrit, Henrik Singmann, Fernando Soler-Toscano, and Hector Zenil. Algorithmic complexity for psychology: a user-friendly implementation of the coding theorem method. *Behavior research methods*, 48(1):314–329, 2016.

[11] Hector Zenil, Fernando Soler-Toscano, Narsis A Kiani, Santiago Hernández-Orozco, and Antonio Rueda-Toicen. A decomposition method for global evaluation of Shannon entropy and local estimations of algorithmic complexity. *arXiv preprint arXiv:1609.00110*, 2016.

[12] Charles H. Bennett. *Logical Depth and Physical Complexity*. IBM Research, 1995.

[13] Luis Antunes, Lance Fortnow, Dieter van Melkebeek, and N.V. Vinodchandran. Computational depth: Concept and applications. *Theoretical Computer Science*, 354(3):391–404, 2006. Foundations of Computation Theory (FCT 2003).

[14] Alfréd Rényi. On measures of entropy and information. In *Proceedings of the fourth Berkeley symposium on mathematical statistics and probability, volume 1: contributions to the theory of statistics*, volume 4, pages 547–562. University of California Press, 1961.