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 Enrique Sánchez
Buenos Aires, 2025

ESTUDIO DE COMPLEJIDAD EN SECUENCIAS DE AMINOÁCIDOS DE PROTEÍNAS

Motivados por la necesidad de explorar patrones estructurales y niveles de aleatoriedad en secuencias biológicas; en este trabajo se presenta el diseño e implementación de una herramienta para el análisis de la complejidad algorítmica de secuencias de aminoácidos de proteínas. Además, se implementaron diversas medidas de complejidad; Icalc, discrepancia, discrepancia en bloque, entropía de Shannon, entropía de segundo orden, compresión basada en gzip y aproximaciones de Kolmogorov y de Bennett. Estas medidas se aplicaron a datos, en formato FASTA, de proteínas naturales obtenidas de UniProt, así como a distintas variantes sintéticas generadas a partir de alterar las secuencias de este dataset. Dichas variantes fueron secuencias *shuffled* (reordenamiento aleatorio de los aminoácidos), *random* (regeneración aleatoria uniforme sobre el alfabeto de aminoácidos), *sorted* (ordenadas alfabéticamente) y *single character* (reemplazadas por un único carácter repetido). La herramienta desarrollada permite cargar conjuntos de secuencias, aplicarles funciones de complejidad y comparar los resultados entre métodos. Este enfoque contribuye a una mejor comprensión del contenido informacional de las secuencias biológicas y ofrece un marco flexible para futuros análisis. Los resultados obtenidos al aplicar esta herramienta sobre el dataset de UniProt refuerzan la hipótesis de que las secuencias de aminoácidos que conforman proteínas que se encuentran en la naturaleza son, en su mayoría, indistinguibles de secuencias sintéticas generadas al azar, siempre y cuando estas conserven la misma distribución de aminoácidos que las proteínas naturales.

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

STUDY OF COMPLEXITY IN PROTEIN AMINO ACID SEQUENCES

Motivated by the need to explore structural patterns and levels of randomness in biological sequences; this work presents the design and implementation of a tool for the analysis of the algorithmic complexity of protein amino acid sequences. In addition, various complexity measures were implemented: Icalc, discrepancy, block discrepancy, Shannon entropy, second order entropy, gzip-based compression and approximations of Kolmogorov and Bennett. These measures were applied to a FASTA format dataset of natural proteins obtained from UniProt, as well as to different synthetic variants generated by altering the sequences from this dataset. These variants were *shuffled* sequences (random rearrangement of amino acids), *random* (uniform random regeneration over the amino acid alphabet), *sorted* (alphabetically ordered) and *single character* (replaced by a single repeated character). The developed tool allows loading sets of sequences, applying complexity functions to them and comparing the results across methods. This approach contributes to a better understanding of the informational content of biological sequences and offers a flexible framework for future analyses. The results obtained when applying this tool to the UniProt dataset reinforce the hypothesis that the amino acid sequences that form proteins found in nature are, for the most part, indistinguishable from synthetic sequences generated at random, as long as they preserve the same amino acid distribution as natural proteins.

Keywords: Complexity, Amino Acid Sequences, Icalc, Discrepancy, Bennett, Kolmogorov, Entropy.

A mi papá.

Índice general

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 frecuentes de observar[1]. Dentro del ámbito de la biología 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 [2, 3]. 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 aminoácidos; si sólo tomamos los 20 aminoácidos más conocidos, una secuencia de largo n tiene 20^n combinaciones posibles. Una posible manera de aproximarse a la respuesta sería reformulando la pregunta a si las proteínas que conocemos que se encuentran en la naturaleza son realmente indistinguibles de secuencias de aminoácidos generadas al azar o simplemente no contamos con una función que permita distinguirlas. Para ello, nos basaremos en conceptos ligados al azar y a la complejidad.

1.2. Azar y Complejidad

El concepto de azar está íntimamente relacionado con el de complejidad [4]. Esta última depende del contexto y de la información disponible para los agentes, y es allí donde entra en juego el azar. En términos de información esperada, un mensaje que confirma la ocurrencia de un evento con probabilidad del 100 % no aporta nueva información, y por ende, carece de complejidad. En cambio, un mensaje que anuncia un evento poco probable contiene más información y, por lo tanto, mayor complejidad que si ese mismo evento fuera más esperable. Por ejemplo, en el caso de un *stream* binario (una fuente de 1 y 0), la afirmación de que el próximo bit será un 1 resulta más informativa si la probabilidad de que ocurra es del 30 % en lugar del 70 %. A lo largo de la historia de la computación se han desarrollado múltiples esfuerzos por precisar el concepto de complejidad. A continuación, se presentan algunos hitos clave en ese proceso.

Una medida de complejidad para los elementos de un conjunto es una función que asigna a cada elemento del conjunto un valor, de forma tal que la mayoría de los elementos resulten ser altamente complejos. A mediados de 1960 Kolmogorov definió una medida de complejidad de palabras, hoy conocida como *complejidad de Kolmogorov*, que asocia alta complejidad con la ausencia de regularidad, es decir, con la falta de patrones en su formación[5].

En los años 70, Gregory Chaitin propuso una variante de esta medida, denominada

complejidad de Chaitin-Kolmogorov, interpretándola como una cantidad de información, con propiedades que coinciden con la noción de entropía de la información de Shannon [6]. Esta variante permitió definir aleatoriedad para secuencias infinitas: una secuencia es aleatoria si todos sus prefijos tienen complejidad máxima. En otras palabras, cada segmento inicial contiene la máxima cantidad posible de información. Esto implica que la única forma de describir tales segmentos es hacerlo de manera explícita, sin compresión posible.

Aunque la complejidad de Chaitin-Kolmogorov constituye una herramienta teórica poderosa, su naturaleza no computable impide su aplicación práctica directa. Por esta razón, se han propuesto diversas alternativas que buscan capturar la idea de complejidad mediante diferentes métodos de descripción, usualmente midiendo la longitud mínima de tales descripciones.

En este contexto, durante los años 70, Lempel y Ziv introdujeron una medida computable de complejidad que combina un enfoque procedimental con propiedades combinatorias de las palabras, conocida como *complejidad de Lempel-Ziv*[7]. Esta medida puede calcularse de manera eficiente mediante un algoritmo que opera en tiempo y espacio lineal respecto del tamaño de la entrada. No obstante, presenta una limitación teórica significativa: existen infinitas palabras cuya complejidad de Lempel-Ziv es menor que cualquier valor prefijado, lo cual contrasta con el comportamiento de la complejidad de Chaitin-Kolmogorov, en la que casi todas las palabras largas tienen alta complejidad. Además, la definición de Lempel-Ziv entrelaza características combinatorias de las palabras con un procedimiento de análisis secuencial, lo que introduce una dependencia del orden en que la información es procesada. Esta característica complica el análisis formal, ya que impide separar claramente la estructura interna de la palabra del mecanismo utilizado para evaluarla, dificultando así el desarrollo de resultados generales y la demostración rigurosa de propiedades teóricas.

La medida de complejidad Icalc presentada en el trabajo de Becher y Heiber[8] ofrece una alternativa computable basada en el conteo de repeticiones. En esta formulación, las secuencias más simples son aquellas que consisten en una única racha de símbolos, mientras que las más complejas son las secuencias de De Bruijn[1]. La mayoría de las secuencias, sin embargo, poseen alta complejidad bajo esta medida.

1.3. Motivación

El estudio de la complejidad en secuencias de aminoácidos se vincula directamente con el origen y el funcionamiento de la vida. Las secuencias de ADN se traducen en cadenas de aminoácidos que, en apariencia, pueden parecer aleatorias. Sin embargo, contienen una complejidad intrínseca que permite la formación de estructuras fundamentales para la vida. Actualmente no contamos con métodos simples que permitan diferenciar de manera clara las secuencias que dan lugar a proteínas funcionales de aquellas que no presentan ninguna función biológica[9]. Esta es la motivación de este trabajo: encontrar alguna medida de complejidad que nos permita discernir entre secuencias de aminoácidos que codifican proteínas funcionales y aquellas que no. El objetivo final es sentar un precedente para cualquier interesado en esta área y proveer una herramienta simple y elegante para el estudio de las secuencias que rigen nuestras vidas.

[1] En matemáticas combinatorias, una secuencia de De Brujin de orden n en un alfabeto de tamaño A, es una secuencia cíclica en donde canda n-secuencia posible en A ocurre exactamente una vez como subcadena.

1.4. 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. Esta tarea, se desglosa en los siguientes subobjetivos específicos (OE):

OE1. Obtener y generar los sets 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

- OE1.a. obtener secuencias de aminoácidos naturales de fuentes de datos públicas;
- OE1.b. generar secuencias de control aleatorias, tanto con distribución uniforme de aminoácidos (*random*) como manteniendo la distribución natural de ellos (*shuffled*) y secuencias de control no aleatorias, manteniendo un mismo aminoácido para toda la secuencia (*single character*) y con los aminoácidos ordenados alfabéticamente (*sorted*).

OE2. Implementar distintas funciones de complejidad para que puedan ser aplicadas a secuencias de aminoácidos y analizar los resultados obtenidos al evaluarlas en las fuentes de datos del punto anterior.

OE3. Desarrollar una herramienta que permita el cálculo y el análisis de diferentes funciones de complejidad aplicadas a fuentes de datos de secuencias de aminoácidos.

2. LA HERRAMIENTA Y METODOLOGÍA

En el marco de esta tesis y con el fin de satisfacer el OE3., se desarrolló la herramienta *Protein Complexity Study Toolkit*, la cual se encuentra subida a GitHub[10]. Esta consiste en un conjunto de funciones implementadas en un proyecto escrito en `Python`, especialmente diseñadas para el estudio de la complejidad en secuencias de aminoácidos. La elección de `Python` se fundamenta en su amplia adopción en la comunidad científica, su versatilidad multiplataforma y su capacidad para facilitar la escritura de código legible, mantenible y escalable; en comparación con lenguajes de más bajo nivel. Asimismo, `Python` dispone de una extensa colección de bibliotecas que permiten su integración con otros lenguajes y herramientas. En particular, en este trabajo se emplea la biblioteca `rpy2` para ejecutar medidas de complejidad originalmente implementadas en `R`.

La herramienta *Protein Complexity Study Toolkit* fue utilizada para generar todos los conjuntos de datos de control empleados en el estudio, así como para aplicar sobre ellos las distintas medidas de complejidad analizadas. La documentación completa de la herramienta se encuentra anexada en la sección 5. A continuación, se describe el procedimiento mediante el cual se generaron los distintos conjuntos de datos de control, utilizando las funciones provistas por dicha herramienta.

2.1. Obtención del Dataset Original

En el marco del objetivo OE1.a. trabajamos con el dataset *Reviewed (Swiss-Prot)* de **UniProt** disponible en su sitio web[1]. Este 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
DTPTNCPTQVCQIVFNMLDDGSVTMDDVKNTINCDFSKYVPPPPPPKPTPPTPPTPPTPPTPPTPPTPRPV
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
```

[1] **UniProt**. Se utilizó *Reviewed (Swiss-Prot)* (www.uniprot.org/help/downloads) versión 17/08/2024

```
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 [2].

Carácter	**Aminoácido**	**Carácter**	**Aminoácido**
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, que consiste en la equivalencia entre aminoácidos y su representación en base a caracteres.

El dataset *Reviewed (Swiss-Prot)* fue filtrado para eliminar aquellas entradas cuya longitud es inferior a 50, con el objetivo de descartar polipéptidos que también se encuentran en este conjunto y que, por tratarse de secuencias simples y de escasa relevancia para nuestro trabajo, no resultan de interés para el presente análisis. La atención se centra en estructuras más extensas y con mayor complejidad. Posteriormente, las secuencias fueron ordenadas de menor a mayor longitud. Esta decisión responde a que las medidas de complejidad se calculan de forma secuencial, comenzando desde el inicio del archivo, y su costo computacional tiende a incrementarse con la longitud de la secuencia. Para la experimentación, fue deseable obtener resultados preliminares lo más rápido posible. Además, esta organización resulta coherente con los criterios adoptados para la visualización de resultados, dado que las medidas fueron graficadas en función de la longitud de las entradas, haciendo que el ordenamiento inicial se refleje directamente en las representaciones gráficas.

El filtrado realizado consiste en eliminar las secuencias con longitud menor a 50 aminoácidos, resultando en 12 839 secuencias eliminadas de un total de 571 864 (2.245 %).

En la figura 2.1.a se observa la distribución de longitudes (en cantidad de aminoácidos) de las entradas originales del dataset (usp) y el filtrado (usp_f) con outliers incluidos. Aquí puede verse claramente la presencia de secuencias extremadamente largas que afectan la representación. Luego, en la figura 2.1.b se muestra la misma distribución pero sin visualizar los outliers, lo que permite apreciar con mayor claridad la variabilidad y el rango principal de longitudes en ambas versiones del dataset.

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

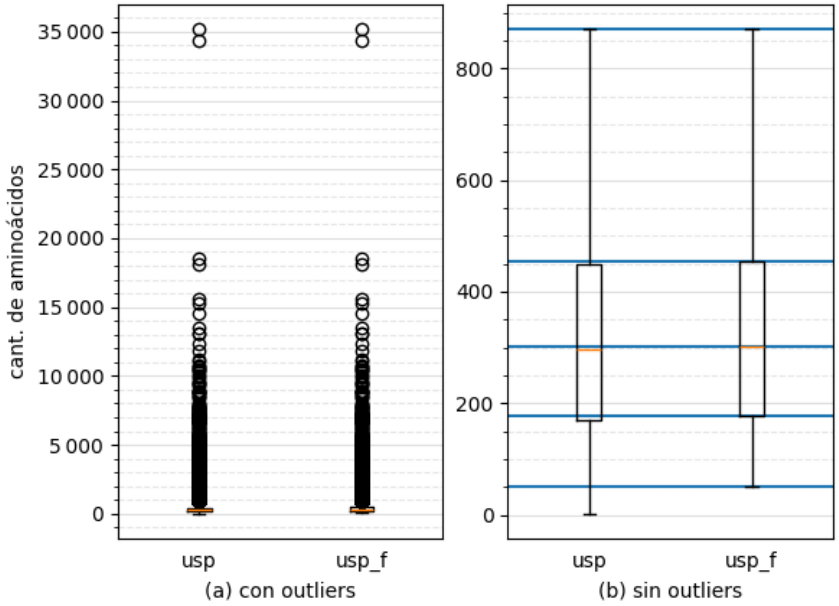

Fig. 2.1: Distribución en largo de las secuencias de aminoácidos del dataset original (`usp`) comparado con la distribución después de remover las secuencias de largo menor a 50 (`usp_f`).Las líneas azules indican la media, bigotes, primer y tercer cuartiles de la distribución `usp_f`. Se visualizan, para ambos casos, con a) todos los outliers y b) removidos todos los outliers.

Para poder hacer esto, se desarrolló dentro de la herramienta la función `filter_to_file (in_file:`*`str`*`, out_file:`*`str`*`, criteria:`*`seq_record -> Bool`*`) -> `*`None`*, la cual permite filtrar un dataset en formato FASTA en función de un criterio definido por el usuario. Donde `in_file` corresponde al archivo de entrada desde que se leerá, `out_file` es el archivo de salida en el que se almacenará el conjunto filtrado, y `criteria` representa una función booleana que define el criterio de selección aplicado a cada secuencia. En caso de cumplirse la condición impuesta sobre una secuencia, la misma será conservada; caso contrario, será descartada. Cabe destacar que esta función, al igual que todas las implementadas en la herramienta, no modifica el archivo de entrada original. De esta manera, el filtrado anteriormente descrito puede realizarse de manera sencilla con el código que se detalla a continuación.

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

Luego, para ordenar el archivo en función del largo de las secuencias de forma ascendente, se desarrolló la función `sort_to_file(in_file:`*`str`*`, out_file:`*`str`*`) -> `*`None`*. Donde `in_file` corresponde al archivo de entrada que se leerá, `out_file` es el archivo de salida en el que se almacenarán una copia ordenada en largo de secuencia del dataset de entrada. Cabe destacar que, dado que no se esperan relaciones entre distintas entradas, el algoritmo de ordenamiento utilizado no es estable. Entonces, con el código a continuación se ordena el archivo `usp_f.fasta` en función del largo de las secuencias en forma ascendente, sobrescribiendo el mismo archivo.

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

Recapitulando, la herramienta *Protein Complexity Study Toolkit* permitió adaptar el dataset según las necesidades del proyecto. Particularmente, se obtuvo de **UniProt** el dataset *Reviewed (Swiss-Prot)*. Al cual se le aplicaron funciones desarrolladas en la herramienta. Primero, se le removieron las entradas de secuencias de largo menor a 50 aminoácidos; segundo, se ordenaron las entradas de forma ascendente en función a su largo de secuencia. Generando, así, el archivo `usp_f.fasta` que consiste en el dataset *original* de trabajo.

2.2. Generación de Datasets de Control

Con el fin de satisfacer el OE1.b., en la herramienta *Protein Complexity Study Toolkit* se implementaron las funciones que permiten generar conjuntos de datos de control. Los conjuntos de control que genera por defecto esta herramienta son *shuffled*, *random*, *single character* y *sorted*, aunque otros pueden agregarse fácilmente. Cada uno consiste en:

- *shuffled* Conjunto de datasets generado con diferentes semillas aleatorias, donde a cada entrada del dataset original, se aleatoriza el orden de aminoácidos de su secuencia.
- *random* Conjunto de datasets generado con diferentes semillas aleatorias, donde a cada entrada del dataset original, se le regeneran, con distribución uniforme sobre el alfabeto de aminoácidos, la secuencia.
- *single character* Dataset donde a cada entrada del dataset original se la reemplaza con una secuencia del mismo largo del carácter `A`.
- *sorted* Dataset donde a cada entrada del dataset original se le ordena la secuencia en orden alfabético.

Los conjuntos de datasets *shuffled* y *random* nos permitieron tener contra qué comparar el resultado de complejidad obtenido. Dado que una medida de complejidad por sí sola es poco útil, resultó de interés generar estos conjuntos de datasets de control para poder ver la efectividad de una medida; es decir, una muy buena medida de complejidad debe permitirnos diferenciar el dataset original del *random* y del *shuffled*, mientras que una buena medida de complejidad, debe permitirnos diferenciar los datasets anteriores del *single character* y del *sorted*.

Para esto, 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`). Se generaron 10 iteraciones aleatorias de cada uno para prevenir que una distribución atípica sesgue los resultados conduciendo a conclusiones erróneas.

Para las secuencias *shuffle* se aplicó 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 se ve un ejemplo

de esto, la transformación de las secuencias tras aplicar las funciones *shuffled* y *random* para la secuencia de ejemplo:

```
>Secuencia de ejemplo
CADENAPEPTIDICA
...
```

La función `shuffle_to_file(in_file:str, out_file:str, seed = 1) -> None`. Donde `in_file` corresponde al archivo de entrada que se leerá, `out_file` es el archivo donde se guardarán las versiones *shuffled* de las secuencias de `in_file` y `seed` es la semilla de aleatoriedad que se usará para mezclar las secuencias de aminoácidos. Notemos que `seed` es por defecto 1 si no se instancia el parámetro, el rango debe ser de 1 a 99 inclusive, para mantener la coherencia en el nombramiento de archivos de salida que posteriormente serán generados. Por ejemplo, si se aplica esta función sobre un archivo que contiene la secuencia mencionada previamente, utilizando la semilla aleatoria 1, se obtiene la siguiente secuencia resultante:

```
>Secuencia de ejemplo
AICCPAEPEDNAITD
...
```

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 comience con la secuencia de ejemplo anteriormente expuesta es el siguiente:

```
>Secuencia de ejemplo
FWDKESRSPHESAPQ
...
```

Para generar estos dataset de control, de forma fácil y configurable, 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 conjuntos de datos *shuffled* y *random* generados constituyen la base principal sobre la que se realizarán los experimentos, ya que permiten abordar la pregunta central de esta tesis: si es posible distinguir las versiones *shuffled* y *random* de una secuencia original mediante alguna medida de complejidad algorítmica, también se generaron datasets de control adicionales con el objetivo de analizar el comportamiento de dichas medidas en escenarios extremadamente simples. Estos casos incluyen el caso *single character*, donde se reemplaza la secuencia de la entrada por una del carácter `"A"` del mismo largo, y el caso *sorted* que ordena en orden alfabético los caracteres de la secuencia 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 entrada, Secuencia de ejemplo, antes mencionada se encuentra en `in_file`, en `out_file`, se encontrará lo siguiente:

```
>Secuencia de ejemplo
AAAAAAAAAAAAAA
...
```

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:

```
>Secuencia de ejemplo
AAACCDDEEIINPPT
...
```

Recapitulando, en esta sección vimos que la herramienta *Protein Complexity Study Toolkit* cuenta con varias funciones para generar fácilmente datasets de control cómo *shuffled*, *random*, *sorted* y *single character*; estas últimas dos nos permitieron entender mejor el funcionamiento de la medida de complejidad utilizada, viendo como se comportaron en casos triviales cuando las secuencias analizadas fueron de un único carácter o una secuencia de caracteres ordenados, respectivamente. Cabe hacer foco en que, mientras que al caso *random* sólo se lo entendió como varias distribuciones aleatorias uniformes sobre el alfabeto de aminoácidos posibles, para los casos *shuffled* existieron varias interpretaciones. Si bien la intención original al mezclar el orden de los aminoácidos fue la de romper las estructuras que puedan existir con orígenes biológicos, funcionales y evolutivos que tenían las proteínas que se estudian; otra interpretación fue que estas fueron secuencias al azar cuya distribución fue la misma que la de las proteínas estudiadas, la cual está lejos de ser uniforme para todo el alfabeto.

Antes de presentar las medidas y los resultados, en la siguiente sección se explica en profundidad los alcances y conceptos claves sobre la herramienta que se desarrolló y utilizó en este trabajo.

2.3. Detalle 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 sencillo y práctico 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 tanto 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 se trata de una función muy poderosa y versátil, y constituye el principal mecanismo mediante el cual la mayoría de las funciones interactúan con el dataset, ya que, si no lo hacen directamente a través de ella, lo hacen manteniendo el mismo esquema general de parsear el dataset entrada por entrada.

Puntualmente, la herramienta consta de cuatro archivos `.py`, `fasta_utils.py` contiene todo lo vinculado al manejo de las fuentes de datos: 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, en base a 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 de la complejidad[3]; y contiene información útil como nombre, prefijo y extensión para los archivos y la función de complejidad a la que hace referencia la clase. En cuanto a los archivos que se generen de la aplicación de la medida de complejidad a un dataset de datos; tendrán extensión `.txt` para aquellas funciones de complejidad que devuelvan un único valor, y extensión `.csv` para 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 este, se encuentran las funciones de más alto nivel que tienen que ver con la manipulación de varios sets de datos a la vez. Dada una función de medida 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 archivo fasta 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. Lo que permite tener un *feedback* a medida que se van ejecutando la medida de complejidad.

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`;
- `complexity_id` es el identificador de la medida de complejidad que se desea evaluar,

[3] Este indicador puede ser `"i"` para Icalc, `"d"` para Discrepancia, `"dN"` para Discrepancia en bloque `N` ó `"b"` para las medidas del Block decomposition method.

esta puede ser Icalc (`"i"`), Discrepancia(`"d"`), Discrepancia en bloque `N` (`"dN"`) ó las medidas de la Block decomposition method (`"b"`);

- `exp` indica el conjunto de datos de control con el que se desea trabajar, únicamente los *shuffled* (`"s"`) ó únicamente con los *random* (`"r"`), por defecto trabaja con ambos (los datos de control *single character* y *sorted* se manipulan con la variable `control`);

- `gen` indica si deben generarse los datasets de control (*shuffled*, *random*, *single character* y *sorted*). Este parámetro debe ser `True` la primera vez que se trabaja con un nuevo dataset, ya que genera los archivos de control. En ejecuciones posteriores debe ser `False` (valor por defecto), dado que los archivos ya existirán.

- `control` indica si se calcula la complejidad sobre los casos *single character* y *sorted*, por defecto sí los calcula (los conjuntos datos de control *shuffled* y *random* se manipulan con la variable `exp`);

- `quantity` número entero entre 0 y 99 que 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` define el modo de ejecución. El modo `"performance"` optimiza el tiempo de cómputo, calculando todas las medidas antes de escribir los resultados en el archivo correspondiente. Mientras que el modo `"feedback"` escribe los resultados a medida que se calculan, lo que permite monitorear el progreso pero implica una mayor demora debido al acceso constante al sistema de archivos.

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 `Python`, por defecto usa un único núcleo (o thread), esto nos permite aumentar la velocidad de cómputo enormemente. Calcular el resultado de una medida de complejidad sobre un conjunto de datos, 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 conjunto de datos 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*. Y, finalmente, se computarán los archivos *sorted* y *single character* de manera secuencial.

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 *chunks* en función del poder de cómputo disponible, entonces se ejecutará primero `f(A1,B1,C1)` 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,C1)` y `f(A2,B2,C2)` se hayan terminado de computar en su totalidad.

Esta es la herramienta que se utilizará para computar las medidas 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_plotting.py` contiene funciones auxiliares utilizadas por los archivos a continuación, `plotting_boxplot.ipynb` permite graficar distintos boxplots y `plotting.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 medidas de complejidad.

Adicionalmente, la herramienta *Protein Complexity Study Toolkit* está diseñada para que agregar nuevas medidas de complejidad sea lo más sencillo posible. Simplemente, se añade la nueva función de complejidad al archivo `complexity_metrics.py` y se la agrega a la clase `ComplexitySelector` respetando la interfaz existente.

3. MEDIDAS DE COMPLEJIDAD Y SUS RESULTADOS

En esta sección, conforme a lo especificado en el OE2., se explicarán cada una de las medidas de complejidad utilizadas y se verán los resultados de aplicarlas a cada uno de los casos de prueba anteriormente desarrollados, buscando que la medida nos permita diferenciar el dataset original de los casos *shuffled* y *random*. Como el 96.6 % de los datos está por debajo de largo 1 000 inclusive, los gráficos fueron cortados en este punto. Dado que los outliers se extienden hasta el largo 35 213, descartarlos permitió una visualización centrada en donde se encuentran la mayoría de los datos; esto se evidencia más adelante en la figura 3.4. Además, para mejorar la visualización en los gráficos, el eje que permite visualizar el largo de la entrada ha sido fijado en escala logarítmica. Todos los gráficos fueron realizados con el código en `plotting.ipynb` 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 2D, manipular la cantidad de *bins* en cada eje que se ven en el gráfico.

Otra consideración es que decidimos generar 10 dataset *shuffled* y 10 dataset *random*. Para estos casos, se grafican los promedios de cada entrada con sus homólogas y, en los gráficos de puntos, también se grafica el desvío estándar. Lo que observamos en la práctica fue 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

Desarrollada por Becher y Heiber, la denominada *I-complejidad* (abreviada *Icalc*) es una medida de complejidad para cadenas, que se caracteriza por ser computable en tiempo y espacio lineales[8]. Esta medida evalúa la diversidad de subcadenas presentes en una secuencia dada: las cadenas menos complejas son aquellas formadas por repeticiones de un único símbolo, mientras que las más complejas corresponden a secuencias de De Bruijn.

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados[1]:

- `icalc("CADENAPEPTIDICA") = 0.7555555555555556`
- `icalc("AICCPAEPEDNAITD") = 0.7555555555555556`
- `icalc("FWDKESRSPHESAPQ") = 0.8555555555555556`
- `icalc("AAAAAAAAAAAAAAA") = 0.1920485995485996`
- `icalc("AAACCDDEEIINPPT") = 0.7555555555555554`

La implementación en `Python` de esta medida puede verse en el Listing 1.

[1] Si bien el valor obtenido al aplicar Icalc a la secuencia original resulta casi equivalente al de la misma secuencia ordenada, en los resultados generales se observa que esta equivalencia varía en secuencias de mayor longitud. Hipotetizamos que esto se debe a que la secuencia de ejemplo no es representativa, dado que consta de solo 15 caracteres en un alfabeto de 20. Casos similares se presentan con otras medidas,

```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)
```

Listing 1: Implementación de la medida de complejidad Icalc en `Python`.

En esta medida de complejidad, 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 computacional final es de $O(n)$ siendo n el tamaño de la entrada.

Resultados

Al igual que con todas las demás medidas de complejidad presentadas en este trabajo, se ejecuta la función de complejidad Icalc sobre cada secuencia de cada fuente de datos. A continuación, se presentan los gráficos correspondientes, donde se muestra el valor de la complejidad en función del largo de la entrada. Cada punto gris representa el resultado del cálculo sobre el dataset *single character*; cada punto violeta, sobre el caso *sorted*; y los puntos celestes, sobre las secuencias originales. Para el caso *shuffled*, al tratarse de un conjunto compuesto por 10 variantes, cada punto naranja corresponde al promedio de los resultados obtenidos al aplicar la medida sobre dichas variantes. Lo mismo se aplica para el caso *random*, cuyos valores promediados se grafican en rojo. Este mismo esquema de graficación se mantendrá para todas las demás complejidades.

En la figura 3.1 podemos ver que Icalc, en términos generales, se puede observar que decrece en función del largo de la entrada. Las secuencias formadas por una único carácter, *single character*, hacen de piso inferior en valores de complejidad, comenzando en el valor de complejidad 0.08 y tendiendo a cero. Luego, es seguido por los resultados del dataset *sorted* diferenciándose de los del original y el *shuffled* poco antes del valor de largo 100 y diferenciado del caso *random* desde el valor de largo 50, es decir desde el inicio del muestreo. Como es de esperar, a modo de límite superior para el valor de complejidad sobre los casos original y *shuffled* se encuentran los resultados *random*, secuencias con distribución aleatoria uniforme; 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

donde esta secuencia de ejemplo no permite visualizar una diferenciación clara, aunque dicha distinción sí puede realizarse al aplicar la medida sobre el total de los conjuntos de datos.

caso *shuffled*, donde se mantiene la frecuencia de aparición de aminoácidos (pero no su posición original), podemos ver una clara superposición con el caso original, aunque este último tenga una dispersión mayor hacia valores de menor complejidad. Esto indica que para algunas secuencias desordenar sus aminoácidos permite destruir estructuras naturales preexistentes en el caso original.

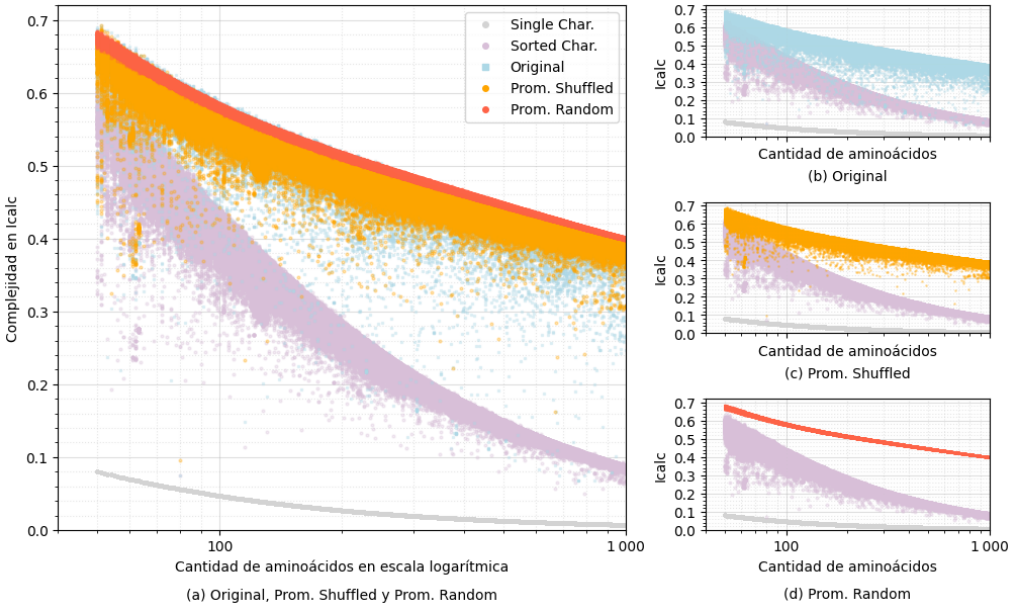

Fig. 3.1: Complejidad en Icalc en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

Dado que los casos *single character* y *sorted* se diferencian claramente del resto, ya no se hará foco en sus resultados de complejidad. Cabe recordar que dichas secuencias funcionan como valores de referencia, útiles para comprender el comportamiento general de cada medida. A continuación, el análisis se centrará en la relación entre los resultados obtenidos para el dataset original y aquellos correspondientes a los conjuntos *shuffled* y *random*. Para ello, se utilizarán gráficos que representen la razón entre los valores del caso original y los promedios obtenidos a partir de las 10 variantes *shuffled* y *random*, respectivamente. Lo mismo se considera para todas las demás complejidades estudiadas. Esto implica que, si la mayoría de los valores se encuentran alejados del 1, la medida nos permite diferenciar con claridad los casos involucrados en la razón; pero, si están sobre la línea del 1, se concluye lo contrario.

Entonces, para visualizar mejor si se logra diferenciar o no el caso original del *shuffled* o del *random*, en el gráfico de la figura 3.2 se grafica la razón entre estos. Particularmente, para el caso *random* 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 línea del 1. También, observamos que en ambos casos abundan los outliers que se encuentran alejados del 1 en todos los largos de entrada. En

particular, estos outliers corresponden a los valores que se ubican por debajo de la línea del valor 0.977 en el caso *shuffled* y por debajo el valor 0.911 para el caso *random*; dado que si se representaran estos resultados como boxplot, en estos valores se encontraría el límite del bigote.

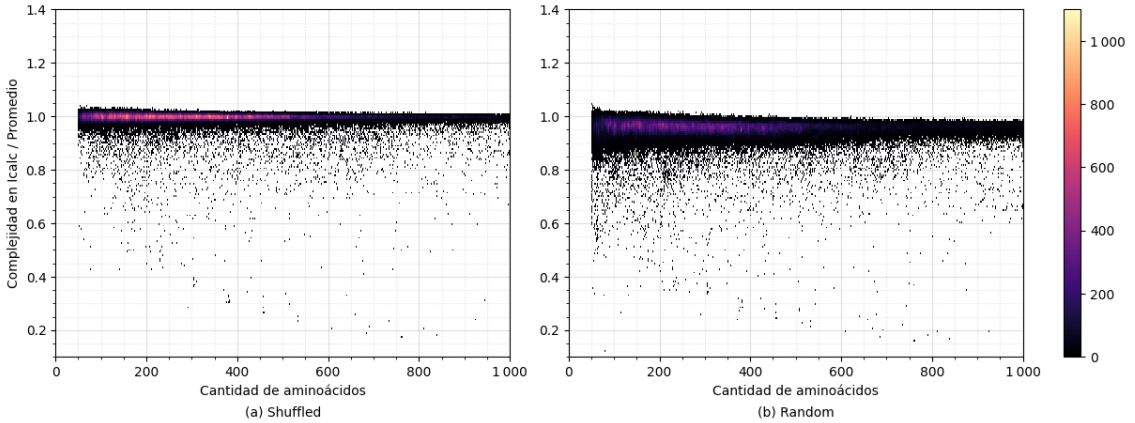

Fig. 3.2: Razón entre la complejidad en Icalc de las secuencias originales y el promedio de complejidad de los casos a) *shuffled* y b) *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

En líneas generales, a partir de los resultados obtenidos se puede evidenciar que Icalc permite notar una diferenciación del caso original con el caso *random* pero no así con el caso *shuffled*. Teniendo en cuenta que existe una pequeña superposición entre los casos original *shuffled* y *random* y existen abundantes outliers del caso original que puntuaron muy por debajo de las otras 2 variantes en esta medida. Lo que indicaría que hay secuencias cuya mayor complejidad estructural puede distinguirse con la medida Icalc, la cual se ve reducida en el conjunto de datos *shuffled* al alterar el orden de los aminoácidos; pero, esta propiedad no es general, pues sucede para algunos casos puntuales y no es mayoritario.

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[11]. 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. Esta medida no está acotada superiormente.

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 $\Sigma \to \mathbb{Z}^*$ 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($\underline{111110}$) = 5
- discrepancia($\underline{110110}$) = 3

- discrepancia(1$\underline{00100}$) = 3

La lógica de la implementación se basa en el algoritmo de Kadane[2]. Para el cual, dado un arreglo de números enteros, permite hallar en tiempo $O(n)$ la sublista contigua cuya suma de elementos es máxima. A modo de comparación, un enfoque *naïve* requeriría un costo de $O(n^2)$ al evaluar todas las combinaciones posibles, lo cual pone de manifiesto la eficiencia y relevancia de Kadane.

Aplicando esta idea al problema de la discrepancia, consideramos una palabra w sobre el alfabeto Σ y dos símbolos arbitrarios $a, c \in \Sigma$. Interpretamos a cada elemento de w como 1 si es igual a a, -1 si es igual a c y 0 en cualquier otro caso. Entonces sobre la lista w se puede calcular $\text{máx}_{s \in S}(|s|_a - |s|_c)$, que implícitamente requiere que la sublista s tenga la mayor cantidad de apariciones de a y la menor cantidad de apariciones de c.

Repitiendo este procedimiento para todas las posibles combinaciones (a, c) del alfabeto Σ y seleccionando el máximo entre los resultados obtenidos, se define la función de discrepancia sobre w.

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados:

- `discrepancy("CADENAPEPTIDICA") = 3`
- `discrepancy("AICCPAEPEDNAITD") = 3`
- `discrepancy("FWDKESRSPHESAPQ") = 3`
- `discrepancy("AAAAAAAAAAAAAAA") = 15`
- `discrepancy("AAACCDDEEIINPPT") = 3`

Notemos que estos resultados son consistentes con los que posteriormente se ven en la experimentación: la discrepancia empieza a puntuar más alto con secuencias más largas. Considerando que tenemos un alfabeto de 20 caracteres, una secuencia de largo 15 resulta muy corta para esta medida.

La implementación en `Python` de esta medida puede verse en el Listing 2.

En cuanto a la complejidad computacional 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 $O(n * k^2) = O(n * 20^2) = O(400n) = O(n)$.

[2] Una explicación detallada e implementación puede encontrarse en: www.geeksforgeeks.org/largest-sum-contiguous-subarray/

```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
```

Listing 2: Implementación de la medida de complejidad Discrepancia en `Python`.

Resultados

De la misma forma que se hizo anteriormente, y como se hace a lo largo de todo este trabajo para cada medida de complejidad, con el fin de analizar el desempeño de esta medida, en la figura 3.3.a se grafican los resultados de la medida de complejidad en función del largo de la secuencia de aminoácidos para el dataset original, *shuffled* y *random*; dado que hay un solapamiento importante entre estas, se grafican de forma aislada en las subfiguras 3.3.b, .c y .d respectivamente los casos original, *shuffled* y *random*.

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 a la vez de límite superior para el valor de complejidad para esta complejidad sin tener ningún tipo de superposición con los demás datasets, mientras que el caso *random* hace a la vez de límite inferior para el valor de complejidad, 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 de visualización. A modo de ejemplo en la figura 3.4.b se ve que en el caso *random* aumenta el desvío estándar en función del largo de las entradas (cantidad de aminoácidos); mientras que en la figura 3.4.a para el caso *shuffled* el desvío estándar es tan bajo que no es posible notarlo sin importar el largo de las entradas. Este caso *random* resulta ser una excepción particular porque para todas las demás medidas de complejidad, tanto en los casos *shuffled* como *random*, el desvío estándar es tan bajo que no resulta apreciable en los gráficos, sin importar el largo de secuencia hasta el cual se grafique.

De forma homóloga a como se hizo en el caso del Icalc y como se hace a lo largo de este trabajo para cada medida, con el fin de ver mejor si existe una diferenciación entre casos;

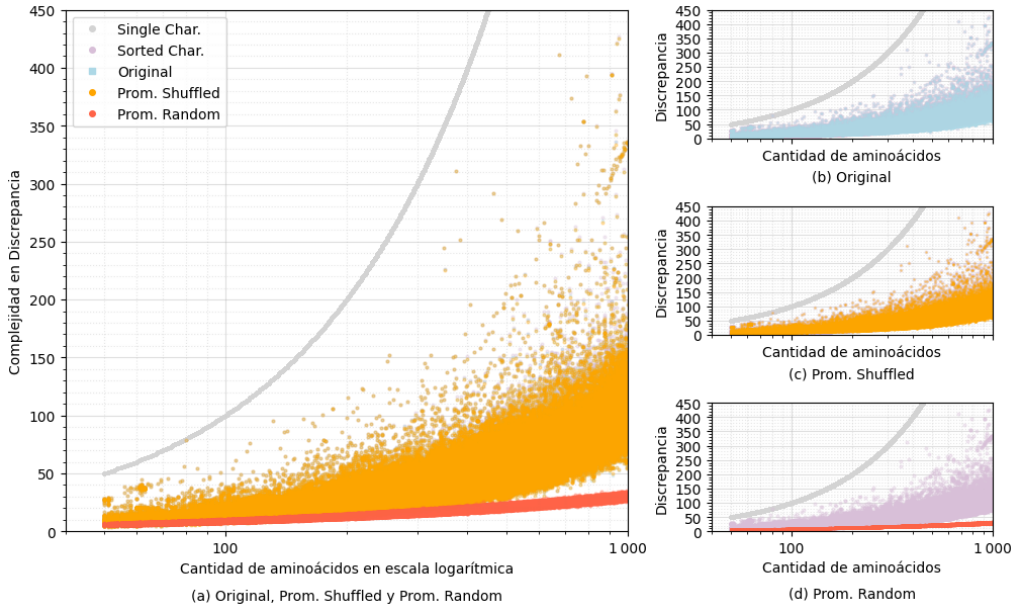

Fig. 3.3: Complejidad en Discrepancia en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

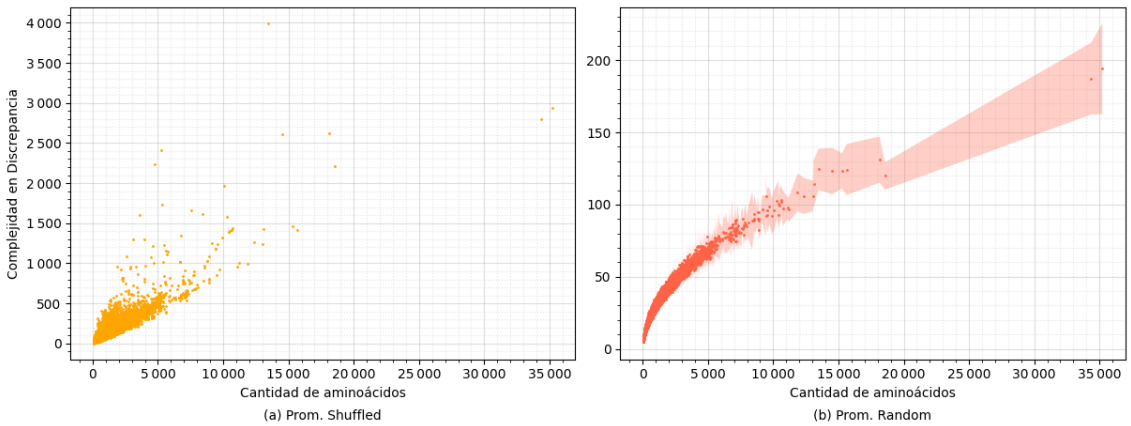

Fig. 3.4: Complejidad en Discrepancia en función del largo de la entrada para el promedio de los resultados a) *shuffled* y b) *random*, en escala lineal. Con desvío estándar mostrado como el sombreado del mismo color de los puntos. Para la totalidad del dataset `usp_f`, considerando outliers.

en el gráfico de la figura 3.5 se grafica la razón entre la Discrepancia en el caso original y la misma con el promedio de los resultados del caso *shuffled*. Podemos ver que hay una gran concentración en torno a la línea del valor 1, mientras que hay mayor dispersión al principio (menor largo) y una menor sobre el final (mayor largo). Mientras que en la figura 3.6 se puede ver que en el caso homólogo *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 valor 1.

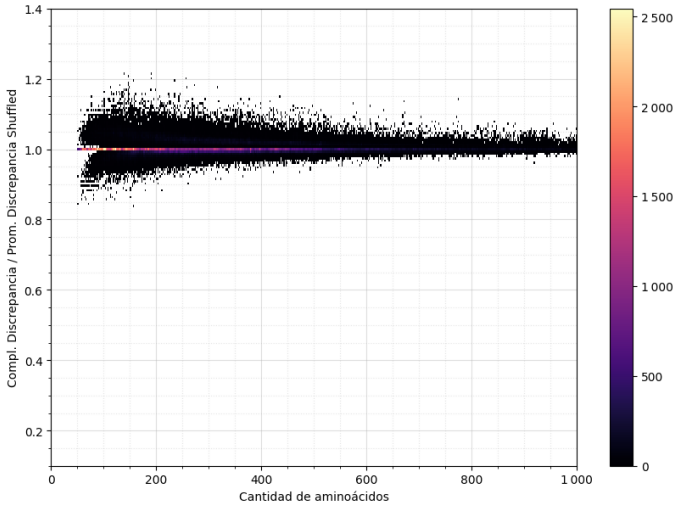

Fig. 3.5: Razón entre la complejidad en Discrepancia de las secuencias originales y el promedio de complejidad del caso *shuffled*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

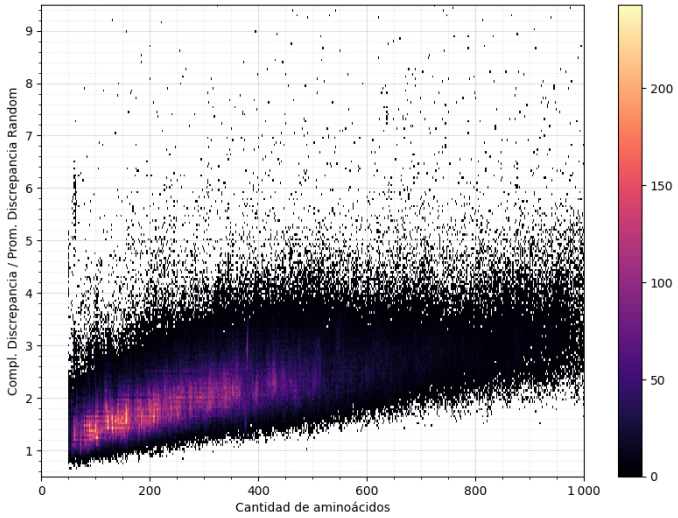

Fig. 3.6: Razón entre la complejidad en Discrepancia de las secuencias originales y el promedio de complejidad del caso *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.05 en el eje y.

A modo de síntesis, la medida de 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. A modo comparativo, con el fin de diferenciar esta medida con la siguiente, de ahora en más nos referiremos a la medida de discrepancia como Discrepancia *naïve*.

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. Al igual que la discrepancia *naïve* esta medida no está acotada superiormente.

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 $\Sigma \ \rightarrow \mathbb{Z}^*$ 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 *naïve* queda contenido en la Discrepancia en Bloques cuando el bloque es de tamaño 1.

En cuanto a la implementación, se conserva la misma estrategia utilizada en la discrepancia *naïve*: emplear el algoritmo de Kadane para recorrer la secuencia con un único índice, lo que garantiza una complejidad computacional de $O(n)$. Sin embargo, esta estrategia presenta dificultades cuando se adapta al cálculo de la *discrepancia en bloques 2* sobre ciertas secuencias. Esto se evidencia en los siguientes ejemplos:

- buscando apariciones del bloque `AA` en la secuencia `AAA` el resultado debe ser 1 y no 2, por lo que no se debe permitir el solapamiento de bloques;
- por el contrario, al calcular la diferencia entre las apariciones de los bloques `AB` Y `BA` en la secuencia `ABA`, el resultado debe ser $1 - 1 = 0$, lo que requiere permitir el solapamiento de bloques.

Entonces, vimos que existen casos donde debemos permitir el solapamiento y casos donde no, lo que implica que con mantener un único índice no es suficiente. Es necesario emplear al menos dos índices distintos. Por esta razón, en el algoritmo implementado se introducen las variables `pos_index` y `neg_index`.

Recordemos que, al utilizar una adaptación del algoritmo de Kadane, se compara cada par de símbolos posibles del alfabeto, considerando todas sus combinaciones. En este contexto, cada índice se actualiza en función de las apariciones de los símbolos que le corresponden: `pos_index` para aquellos que incrementan la suma y `neg_index` para aquellos que la decrementan.

A medida que avanza el índice principal del algoritmo, se cuenta la aparición de un bloque positivo únicamente si la posición actual supera el valor almacenado en `pos_index`, actualizándolo a continuación. Lo mismo ocurre para los bloques negativos con `neg_index`. Este mecanismo impide el solapamiento entre bloques del mismo tipo (ya sean positivos o negativos), pero permite el solapamiento entre bloques de distinto tipo, lo cual es coherente con la lógica de cómputo de la discrepancia en bloques.

Por ejemplo, aplicarle discrepancia en bloque 2 a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados:

- `discrepancy("CADENAPEPTIDICA", 2) = 2`
- `discrepancy("AICCPAEPEDNAITD", 2) = 2`

- `discrepancy("FWDKESRSPHESAPQ", 2) = 2`
- `discrepancy("AAAAAAAAAAAAAAA", 2) = 7`
- `discrepancy("AAACCDDEEIINPPT", 2) = 1`

Mientras que, al aplicar la discrepancia en bloque 3 para los mismos casos se obtienen los resultados:

- `discrepancy("CADENAPEPTIDICA", 3) = 1`
- `discrepancy("AICCPAEPEDNAITD", 3) = 1`
- `discrepancy("FWDKESRSPHESAPQ", 3) = 1`
- `discrepancy("AAAAAAAAAAAAAAA", 3) = 5`
- `discrepancy("AAACCDDEEIINPPT", 3) = 1`

La implementación en `Python` de esta medida puede verse en el Listing 3.

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 que 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 aumento de la constante b hace que en la práctica el aumento en tiempo sea 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.

```python
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
```

Listing 3: Implementación de la medida de complejidad Discrepancia en Bloque en `Python`.

Resultados

Debido al alto costo temporal práctico de la medida de discrepancia en bloque 2 para el dataset usado en este trabajo; sólo para esta medida, utilizamos una submuestra del dataset `usp_f`, `usp_987` que consiste en todas las entradas de `usp_f` de largo exactamente 300 (del estudio de `usp_f` vimos que la media está alrededor de este número) más otras 2 secuencias que resultaron de interés para el codirector de la presente tesis; ambas se encuentran al principio de `usp_987`, con largo 393 y 317 respectivamente. Estas se verán al final de esta subsección.

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

En cuanto al caso *single character* las entradas 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 aparte en la figura 3.7. En esta figura podemos ver que la mediana 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.

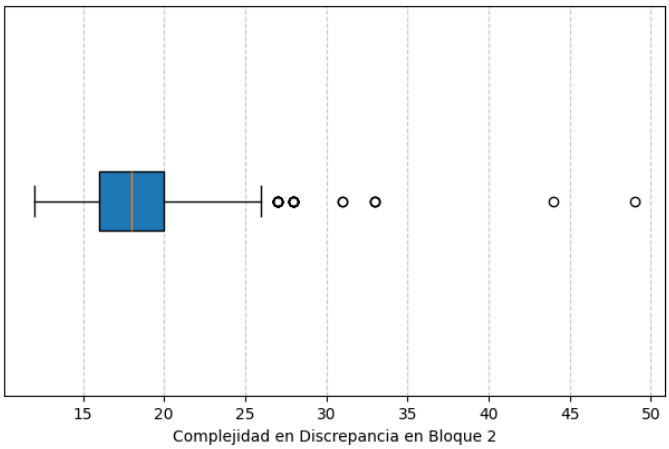

Fig. 3.7: Complejidad en Discrepancia en Bloque 2 para las secuencias del dataset *sorted* de `usp_987`.

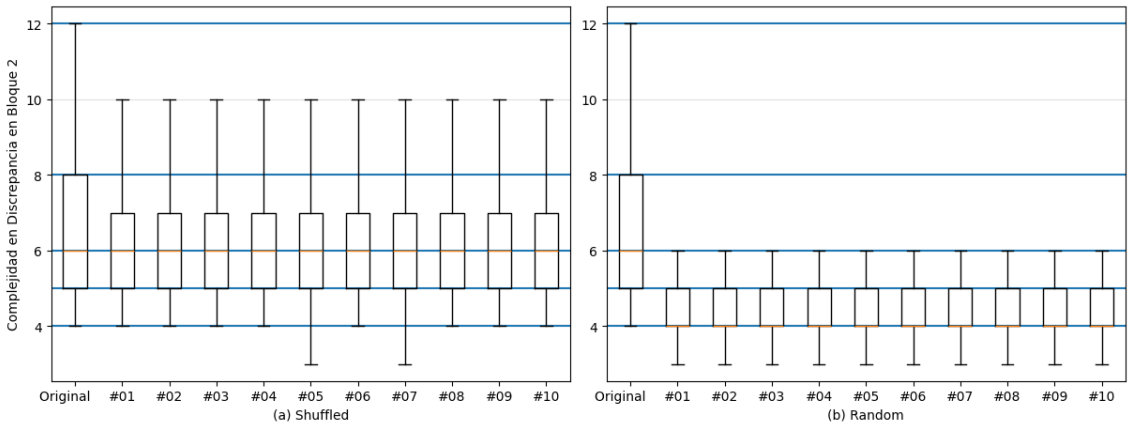

Fig. 3.8: Complejidad en Discrepancia en Bloque 2 para la secuencia original y sus 10 variantes a) *shuffled* y b) *random*. Las líneas azules indican la media, bigotes, primer y tercer cuartiles de la distribución original. En ambas subfiguras se removieron los outliers.

No tiene sentido realizar un gráfico en función de la longitud de la secuencia, dado que, exceptuando 2, todas las entradas tienen largo 300. Por ello, se realizaron un boxplot por cada conjunto de datos analizados, los cuales pueden verse en la figura 3.8. En esta podemos ver que la mediana 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; con un leve desbalance hacia los valores altos. En el caso *shuffled* se comparten las medianas, primer cuartil y (en la mayoría de los casos) bigote inferior con el caso original; si bien el caso original logra bigotes superiores 20 % mayores que las variantes *shuffled* y el tercer cuartil se encuentra por encima de este en el grueso de los datos se encuentra aún en cercana semejanza con el caso original como para poder decir que son diferenciables. Mientras que el caso *random* podemos ver que hay un claro desacople con el original indicando una mejor diferenciación con el caso original

que la que se veía con el caso *shuffled*.

Como se mencionó anteriormente, este dataset cuenta con dos secuencias de interés para el codirector de esta tesis, son las siguientes y estos fueron sus resultados (considerando que para los casos *shuffled* y *random* los resultados se anumeran desde la semilla 1 a la 10):

```
>sp|P04637|P53_HUMAN Cellular tumor antigen p53 OS=Homo sapiens OX=9606
GN=TP53 PE=1 SV=4
```

- Largo: 393
- original: 9
- *shuffled*: 7, 6, 6, 7, 7, 7, 6, 8, 6 y 6
- *random*: 5, 5, 5, 5, 7, 4, 5, 4, 4, y 5
- *sorted*: 22
- *single character*: 196

```
>sp|P25963|IKBA_HUMAN NF-kappa-B inhibitor alpha OS=Homo sapiens OX=9606
GN=NFKBIA PE=1 SV=1
```

- Largo: 317
- original: 7
- *shuffled*: 6, 5, 10, 6, 7, 7, 6, 8, 9 y 7
- *random*: 4, 4, 4, 4, 4, 4, 8, 5, 4 y 6
- *sorted*: 23
- *single character*: 158

Entonces, se ve que los resultados *single character* hacen la vez de límite superior para el valor de complejidad, al igual que con la Discrepancia *naïve*. Cabe notar que la Discrepancia en Bloque 2 permite diferenciar el caso original del *sorted*, lo que no pudo lograrse con la Discrepancia *naïve* donde estos se veían igual que el *shuffled*. Y si bien parece lograr una diferenciación levemente mejor que la Discrepancia *naïve* en los casos *shuffled* y *random*, al ser una versión reducida del dataset no es posible generalizar la conclusión.

3.4. Medidas provistas por OACC

Existe un conjunto de medidas de complejidad provistas por Soler et.al, Gauvrit et. al y Zenil et. al [12, 13, 14]. Las mismas constan de una implementación en `R` de las medidas de *The Online Algorithmic Complexity Calculator* (OACC)[3], esta implementación permite calcular aproximaciones a través del *Block Decomposition Method* (BDM, basado en la Probabilidad Algorítmica[4]) a complejidades teóricas que no son computables como la

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

complejidad algorítmica de Kolmogorov[5] o la profundidad lógica de Bennett[15], además también provee un valor para 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, siendo la primera y principal es el alfabeto. La herramienta OACC, si bien fue adaptada para su uso en este proyecto, no permite trabajar directamente con alfabetos de 20 símbolos, como es el caso del alfabeto de aminoácidos. En su lugar, se emplea la conversión `"256 (utf-8)"`, que consiste en tomar la secuencia representada en el estándar UTF-8 (8-bit Unicode Transformation Format, un formato de codificación de caracteres conforme a Unicode e ISO 10 646) y convertirla a una secuencia binaria; y luego, calcular sobre esta la complejidad mediante el método BDM. Adicionalmente, el cálculo de BDM requiere definir los parámetros `Block Size` ($\mathbb{Z}$, en el rango de 2 a 12) y `Block Overlap` ($\mathbb{Z}$, en el rango de 0 a `Block Size` - 1). En el presente trabajo se optó por utilizar `Block Size` = 12 y `Block Overlap` = 0. Ya que, tras realizar pruebas empíricas, se observó que esta combinación permite obtener resultados de forma más eficiente en términos de tiempo de cómputo en comparación con otras configuraciones posibles.

Es importante remarcar que el BDM calcula aproximaciones de medidas teóricas de complejidad que, por definición, no son computables. Por ello, establecer la superioridad de una configuración de parámetros sobre otra es una tarea compleja y escapa al alcance de esta tesis. A continuación se profundiza sobre cada una de estas medidas 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[16]. Sin embargo, la información aleatoria puede no ser muy útil desde un punto de vista computacional, porque queremos diferenciar secuencias con complejidad estructural biológica de la que no la tienen, no solamente las que son más aleatorias que otras. Los resultados de esta medida son mayores a cero y no tienen cota superior.

Formalmente, un programa p es una descripción de la cadena x si p ejecutado en una máquina de Turing universal U produce x, y se escribe $U(p) = x$. La longitud de la descripción más corta, la complejidad de Kolmogorov, se denota por

$$K(x) := \min_{p}\{\ell(p) : U(p) = x\}$$

donde $\ell(p)$ es la longitud de p medida en bits[15].

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados (nótese que la complejidad en Kormogorov es el primer resultado del cálculo del *block decomposition method*, `bdm()`):

- `bdm("CADENAPEPTIDICA")[0] = 315.1786`
- `bdm("AICCPAEPEDNAITD")[0] = 285.4688`

- `bdm("FWDKESRSPHESAPQ")[0] = 286.0363`
- `bdm("AAAAAAAAAAAAAAA")[0] = 65.5448`
- `bdm("AAACCDDEEIINPPT")[0] = 312.5115`

Resultados

Se grafican los resultados de esta medida en función del largo de la entrada en la figura 3.9, donde se ve que los casos original, *shuffled* y *random* crecen en función del largo de la entrada. Estos tres se separan del caso *sorted* en 200 aproximadamente y junto con *single character* se mantienen constantes en función del largo de la entrada, estando *sorted* siempre por debajo de 1 800 y por encima de 1 400 después del largo de entrada 400; y *single character* entre 60 y 110. 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 elementos del dataset original y *shuffled* se ven como *random* para esta medida.

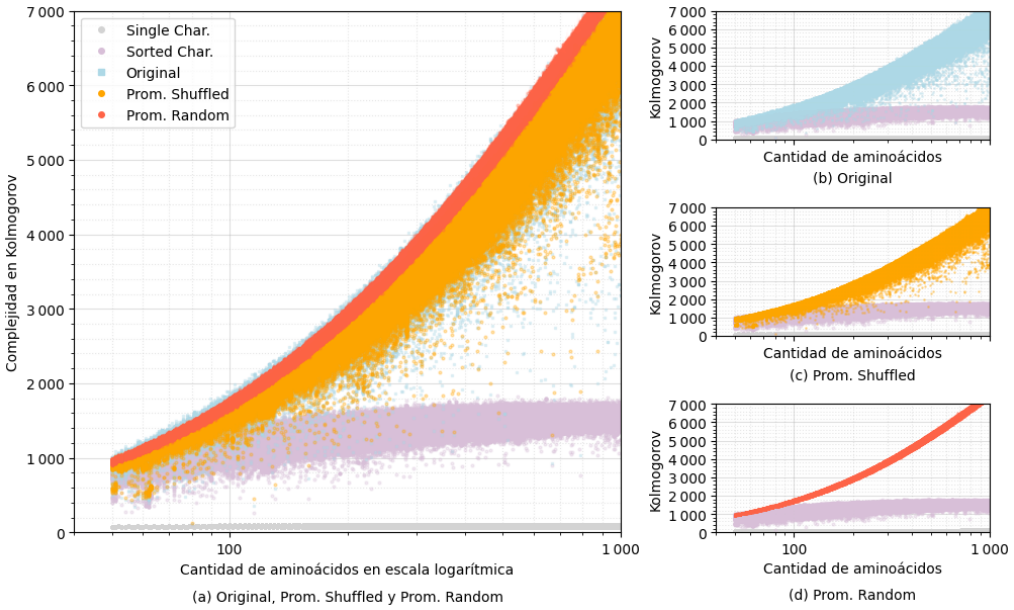

Fig. 3.9: Complejidad en Kolmogorov en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

Para analizar en profundidad la capacidad de disntiguibilidad que provee esta medida, se grafican los resultados de la razón de la complejidad de Kolmogorov medida entre las secuencias originales y las variantes *shuffled* (figura 3.10.a) y *random* (figura 3.10.b). Se observa que la medida de Kolmogorov no permite diferenciar de forma concluyente el caso *shuffled* respecto del original, más allá de la dispersión inicial atribuible al ruido estadístico en secuencias cortas. En el caso *random*, si bien se advierte una tendencia decreciente

en la razón de complejidad, esta diferenciación resulta limitada y no presenta mejoras significativas respecto a las medidas analizadas previamente.

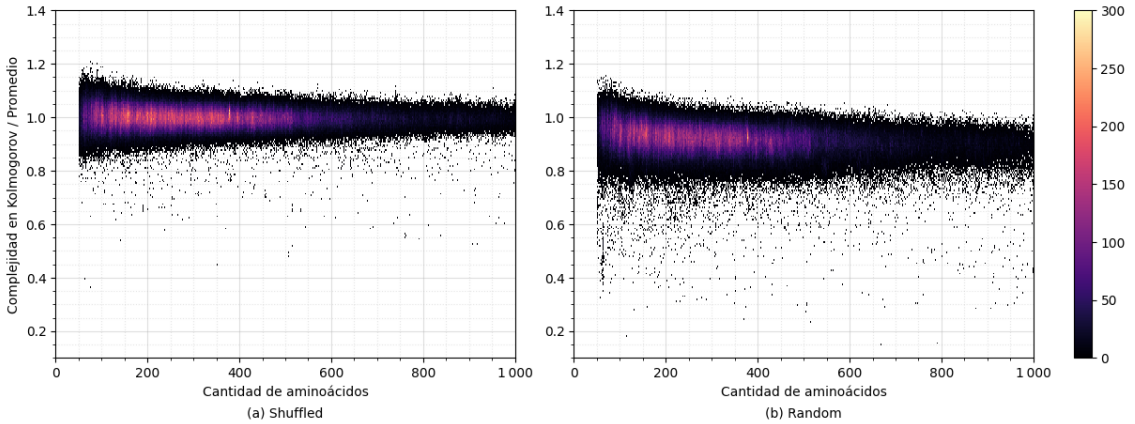

Fig. 3.10: Razón entre la complejidad en Kolmogorov de las secuencias originales y el promedio de complejidad de los casos a) *shuffled* y b) *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

Podemos concluir que el desempeño de esta medida 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 gráficos 3.2 y 3.10. Salvando la diferencia de que Icalc presenta muchos más outliers (puntos aislados por debajo de la aglomeración principal) en ambos casos y que Kolmogorov presenta una mayor dispersión alrededor de la línea del valor 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 cantidad de 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. El resultado de esta medida es mayor a cero y no tiene cota superior.

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*[15].

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados (nótese que la complejidad en Bennett es el segundo resultado del cálculo del *block decomposition method*, `bdm()`):

- `bdm("CADENAPEPTIDICA")[1] = 385.0`

- `bdm("AICCPAEPEDNAITD")[1] = 367.0`
- `bdm("FWDKESRSPHESAPQ")[1] = 358.0`
- `bdm("AAAAAAAAAAAAAAA")[1] = 312.2612`
- `bdm("AAACCDDEEIINPPT")[1] = 368.0`

Resultados

En la figura 3.11 se grafican los resultados de esta medida en función del largo de la entrada, donde se ve que en todos los casos los resultados de complejidad crecen en función del largo de la entrada. El caso *single character* 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 con las medidas de complejidad Icalc y Kolmogorov, se tiene que parte de los casos original y *shuffled* se comportan como *random*, destacando que es posible ver outliers (puntos aislados por fuera de la aglomeración principal) del caso original por encima del caso *random* y el *shuffled*, además de por debajo; esto no sucedió para ninguna de las medidas vistas anteriormente. Mientras que se logra cierta diferenciación entre el caso original y el *random*, esta no se logra con el caso *shuffled*.

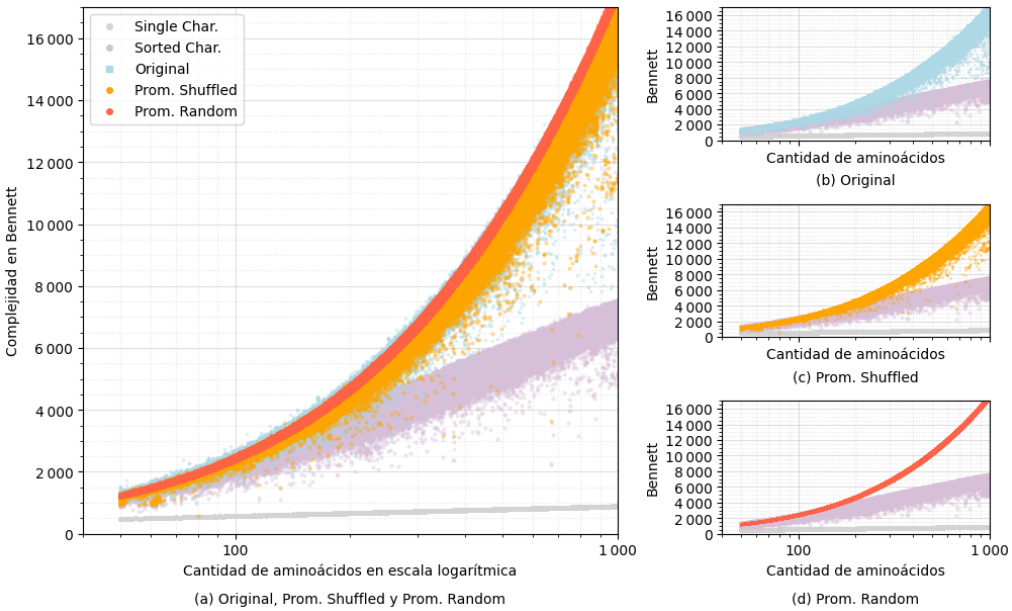

Fig. 3.11: Complejidad en Bennett en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

El gráfico de la figura 3.12 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.

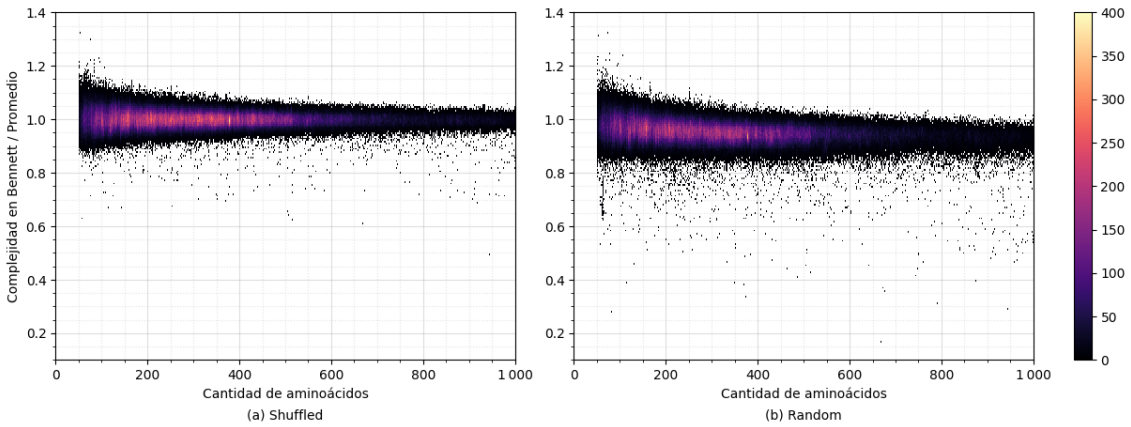

Fig. 3.12: Razón entre la complejidad en Bennett de las secuencias originales y el promedio de complejidad de los casos a) *shuffled* y b) *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

Cabe destacar lo llamativo de la similitud de los resultados entre las medidas Kolmogorov y Bennett; siendo la segunda más sofisticada que la primera (debido a que esta considera el tiempo de cómputo del algoritmo de longitud mínima, mientras que Kolmogorov se centra sólo en la longitud de dicho algoritmo). Al tener en cuenta más factores, se esperarían mejores resultados, es decir, visibilizar una mejor diferenciación entre casos de la que se vio en la práctica.

3.4.3. Shannon entropy (bit(s))

La medida de 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 de información. Estos métodos forman la base de muchos, si no de la mayoría, de los algoritmos de compresión más comúnmente utilizados [14]. El resultado de esta medida es mayor a cero y no tiene cota superior.

De manera informal, la Shannon entropy es una medida de incertidumbre que mide la cantidad promedio de información que se espera obtener al observar el resultado de una fuente aleatoria, considerando la probabilidad de aparición 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 símbolo x_i para todos los i estados posibles.

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados (nótese que la complejidad en Shannon entropy es el tercer resultado del cálculo del *block decomposition method*, `bdm()`):

- `bdm("CADENAPEPTIDICA")[2] = 2.9232`

- `bdm("AICCPAEPEDNAITD")[2] = 2.9232`
- `bdm("FWDKESRSPHESAPQ")[2] = 3.3232`
- `bdm("AAAAAAAAAAAAAAA")[2] = 0.0`
- `bdm("AAACCDDEEIINPPT")[2] = 2.9232`

Resultados

Graficando los resultados de la complejidad Shannon entropy para todos los conjuntos de datos, en la figura 3.13 se ve que la complejidad casi no aumenta después del largo 100; si bien esta medida no tiene cota superior, se estima que esto se debe a que el alfabeto está acotado. El caso *single character* hace a la vez de límite inferior para el valor de complejidad siendo apenas visible por estar tan pegado al cero. Mientras que el caso *random* hace a la vez de límite superior para el valor de complejidad separándose del caso original, *sorted* y el *shuffled*, que son idénticos, luego del largo 400 aproximadamente.

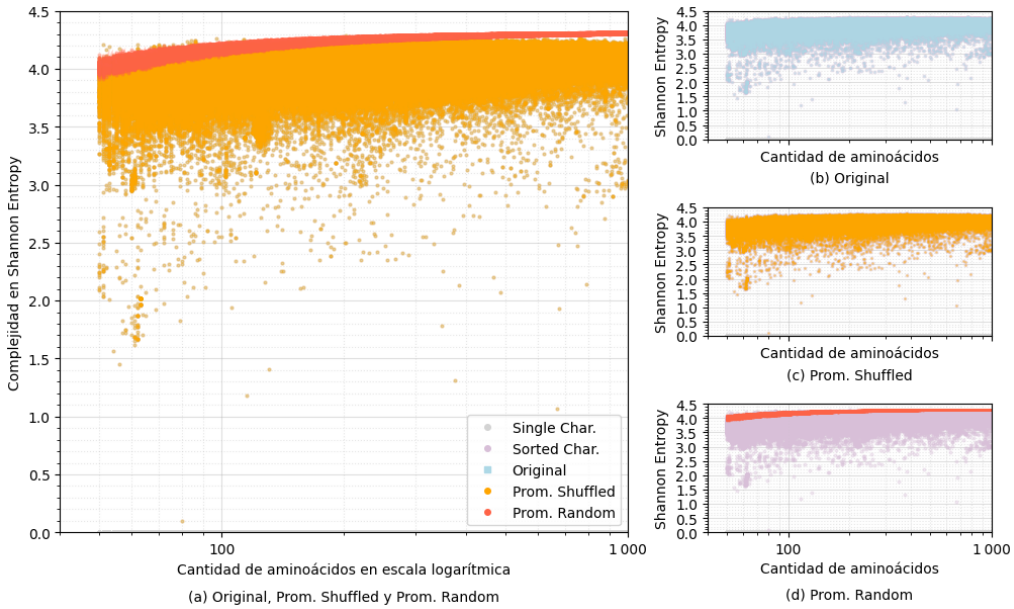

Fig. 3.13: Complejidad en Shannon entropy en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

En el gráfico de la figura 3.14 podemos confirmar lo que vimos anteriormente, el caso *shuffled* se ve como una perfecta línea sobre el valor 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 valor 1, si bien antes del 300 el centro de la distribución está claramente desviada hacia debajo del valor 1.

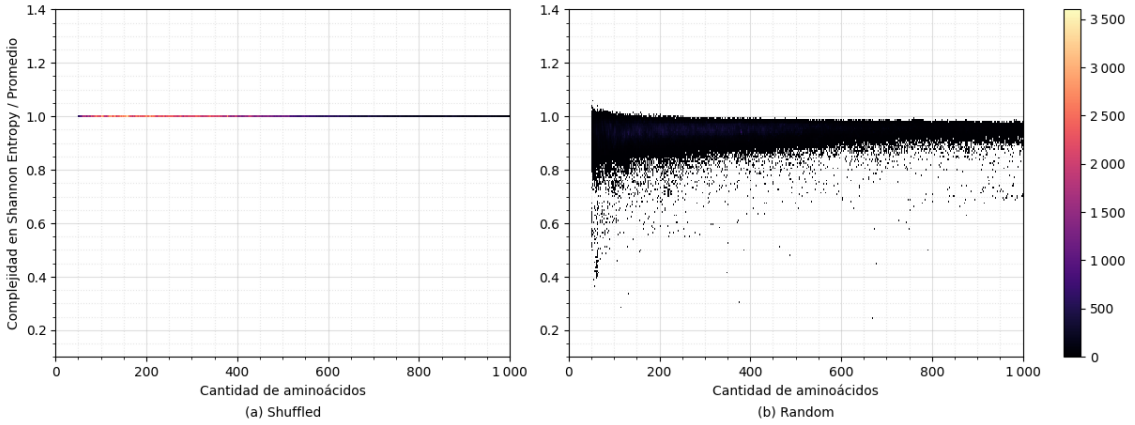

Fig. 3.14: Razón entre la complejidad en Shannon entropy de las secuencias originales y el promedio de complejidad de los casos a) *shuffled* y b) *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

En síntesis, esta medida no nos permitió separar los casos original, *sorted* y *shuffled* que resultaron en valores idénticos para la Shannon entropy. 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. Por esto mismo, se logra una pronta y clara diferenciación entre el caso original y *random* que se separan totalmente luego del valor de cantidad de aminoácidos 400.

3.4.4. Second order entropy (bit(s))

Shannon extendió su definición de entropía vista anteriormente, conocida como entropía de Shannon o entropía de primer orden, a bloques de longitud n, dando lugar a la llamada *entropía en bloques*:

$$H_n = -\sum_{x_1,\ldots,x_n} P(x_1,\ldots,x_n)\,\log_2 P(x_1,\ldots,x_n),$$

donde $P(x_1,\ldots,x_n)$ es la probabilidad conjunta de que aparezca el bloque $(x_1,\ldots,x_n)$ como subcadena de longitud n [17].

Para el caso $n = 2$, esta fórmula se reduce a

$$H_2 = -\sum_{x,y} P(x,y)\,\log_2 P(x,y),$$

que en la OACC se expone con el nombre de *Second order entropy*. Esta medida captura las correlaciones entre pares de símbolos consecutivos, proporcionando una cuantificación más sofisticada de la estructura de la secuencia que la entropía de primer orden. Al igual que con Shannon entropy, el resultado de esta medida es mayor a cero y no tiene cota superior.

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes

resultados (nótese que la complejidad en Second order entropy es el cuarto resultado del cálculo del *block decomposition method*, `bdm()`):

- `bdm("CADENAPEPTIDICA")[3] = 3.6645`
- `bdm("AICCPAEPEDNAITD")[3] = 3.6645`
- `bdm("FWDKESRSPHESAPQ")[3] = 3.6645`
- `bdm("AAAAAAAAAAAAAAA")[3] = 0.0`
- `bdm("AAACCDDEEIINPPT")[3] = 3.6645`

Resultados

En la figura 3.15, donde se grafican los resultados de la complejidad en función del largo de la secuenica, vemos que la complejidad en Second order entropy 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 hacia valores inferiores levemente mayor que el *shuffled*.

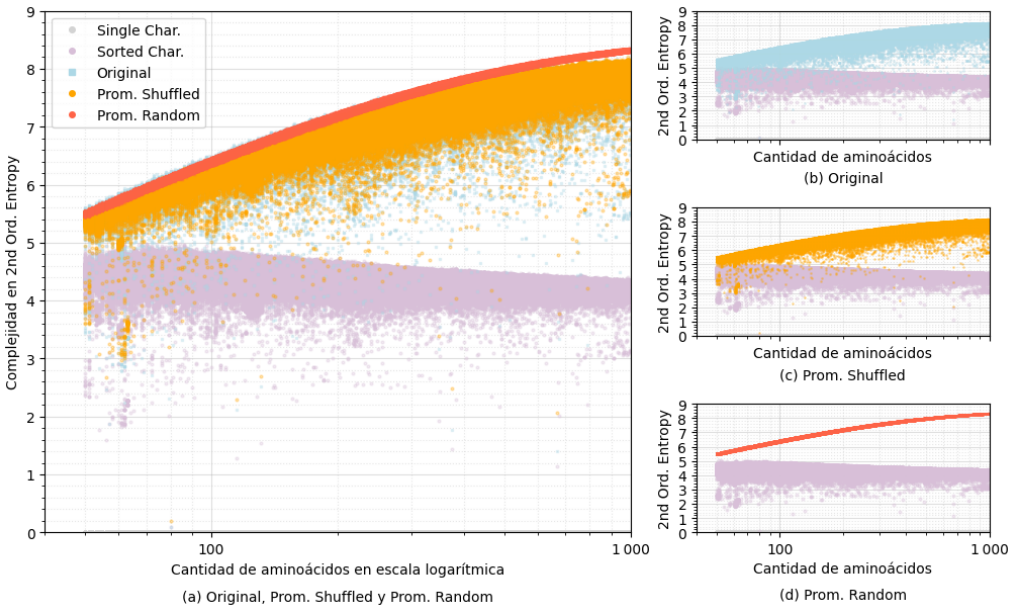

Fig. 3.15: Complejidad en Second order entropy en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

Con el fin de profundizar el análisis, en el gráfico de la figura 3.16 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 (si se graficaran como boxplots, los bigotes llegarían al 0.979 y 0.946 para los casos *shuffled* y *random* respectivamente), que los que se hubieran esperando sólo viendo la figura 3.15 (en las figuras 3.15 y 3.1, Second order entropy e Icalc respectivamente, se ven varios outliers bajo la aglomeración principal, lo que sugiere comportamiento similar: existen algunas secuencias donde hay una buena diferenciación entre el caso original y el *shuffled*; finalmente, la figura 3.16 permite ver que esto no se da para el caso de la complejidad Second order entropy como sí se dio en el caso del Icalc).

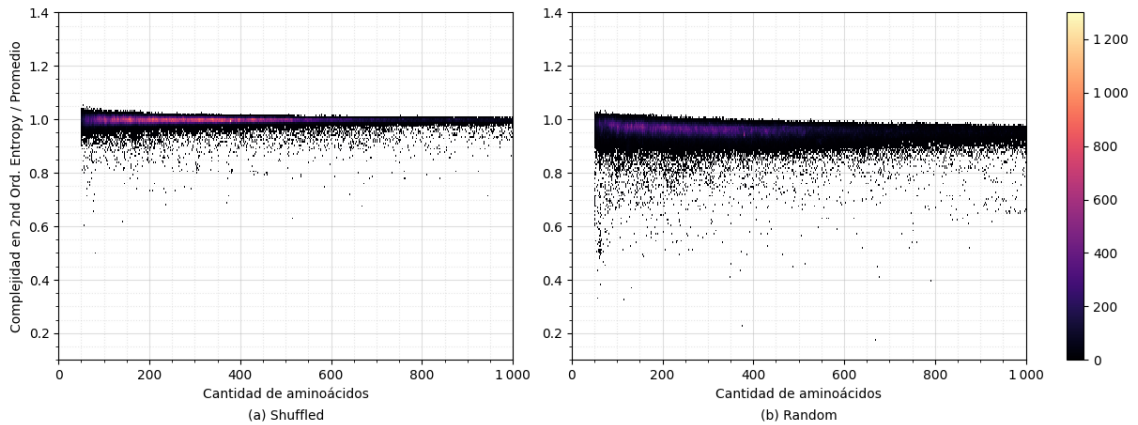

Fig. 3.16: Razón entre la complejidad en Second order entropy de las secuencias originales y el promedio de complejidad de los casos a) *shuffled* y b) *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

En resumen, la medida de complejidad Second order entropy para secuencias mayores al largo 500 permite diferenciar satisfactoriamente el caso original del *random*, como sucedió con medidas anteriores. Pero no es capaz de diferenciar el caso original del *shuffled*, sin importar el largo de las secuencias.

3.4.5. Compression length using gzip (bits)

La medida Compression length using gzip (de ahora en más sólo Compression length) mide cuántos bits se necesitan para representar una secuencia después de ser comprimida por el algoritmo de `gzip`. Puntualmente, `gzip` implementa el algoritmo de ventana deslizante LZ77 combinado con codificación de Huffman [18]. 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 medida que permite reflejar la redundancia o aleatoriedad de las secuencias[14]. El resultado de esta medida es mayor a cero y no tiene cota superior.

Formalmente, dada una cadena de entrada x, su longitud de compresión con gzip se

define como

$$L_{\text{gzip}}(x) \;=\; 8 \times \left|\text{gzip}(x)\right|_{\text{bytes}},$$

donde gzip(x) representa el flujo de bytes resultante de la compresión y se multiplica por 8 para convertir de bytes a bits.

Por ejemplo, aplicarle esta medida a la Secuencia de ejemplo vista anteriormente, para los casos original, *shuffled*, *random*, *single character* y *sorted*, se obtienen los siguientes resultados (nótese que la complejidad en Compression length es el quinto resultado del cálculo del *block decomposition method*, `bdm()`):

- `bdm("CADENAPEPTIDICA")[4] = 184.0`
- `bdm("AICCPAEPEDNAITD")[4] = 184.0`
- `bdm("FWDKESRSPHESAPQ")[4] = 184.0`
- `bdm("AAAAAAAAAAAAAAA")[4] = 88.0`
- `bdm("AAACCDDEEIINPPT")[4] = 184.0`

Nótese que, al igual que sucedió con medidas de complejidad anteriores, esta secuencia de ejemplo, si bien permite tener una idea de cómo funciona la medida, no es capaz de demostrar sus capacidades; al tratarse de secuencias cortas, de 15 caracteres para un alfabeto de 20. Resultados y diferenciación significativos se encuentran en secuencias más largas; este punto permite ilustrar y fundamentar por qué se decidió, al principio de este trabajo, descartar las secuencias de largo menor a cincuenta.

Resultados

Graficando los resultados de esta medida en función del largo de las secuencias, en la figura 3.17 se puede ver cómo la complejidad en Compression length 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 600 para el caso *sorted* y a penas logra crecimiento en cuanto el caso *single character*. Este último se encuentra totalmente diferenciado de las demás medidas, mientras que el caso *sorted* se separa del original alrededor del largo 100. Se ve un solapamiento entre los casos original, *shuffled* y *random* hasta aproximadamente el largo 900; lo que indica que para esta medida parte de los dataset original y el promedio *shuffled* se comportan de manera similar al de el promedio *random*. En cuanto al caso *shuffled* este se encuentra solapado con el original, con este último teniendo varios outliers (puntos aislados por debajo del aglomerado principal) por debajo del *shuffled*; sugiriendo 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*.

Para profundizar el análisis, en los gráficos de la figura 3.18 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 del largo 600. Mientras que en el caso *shuffled* la mayoría de los puntos se encuentran en la línea del valor 1, con una buena presencia de outliers por debajo de este, presentes en todos los largos de las secuencias. Para este caso, si se graficaran como boxplot, los límites de los bigotes inferiores estarían en 0.983 y 0.94 para los casos *shuffled* y *random* respectivamente.

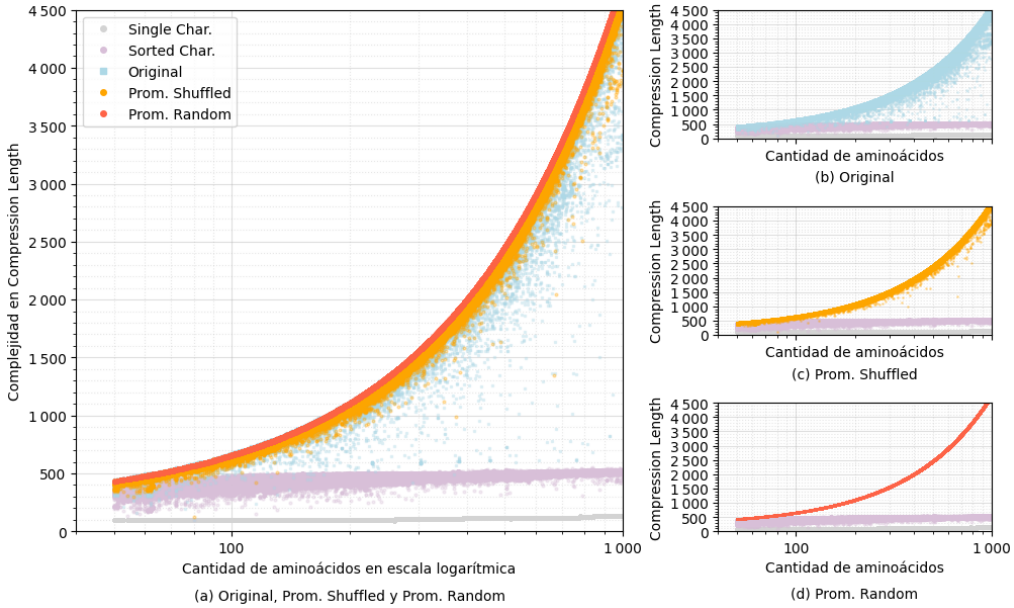

Fig. 3.17: Complejidad en Compression length en función del largo de la entrada. Cada punto celeste representa el resultado de una entrada en el caso original mientras que los naranja y rojo representan el promedio de los 10 resultados para los casos *shuffled* y *random* respectivamente. De fondo, como referencia, se incluyen los resultados *sorted* en violeta y *single character* en gris en todas las subfiguras. Las subfiguras a la derecha presentan los casos b) original, c) *shuffled* y d) *random*; esto permite visualizar estas tres últimas clases sin el solapamiento natural observado.

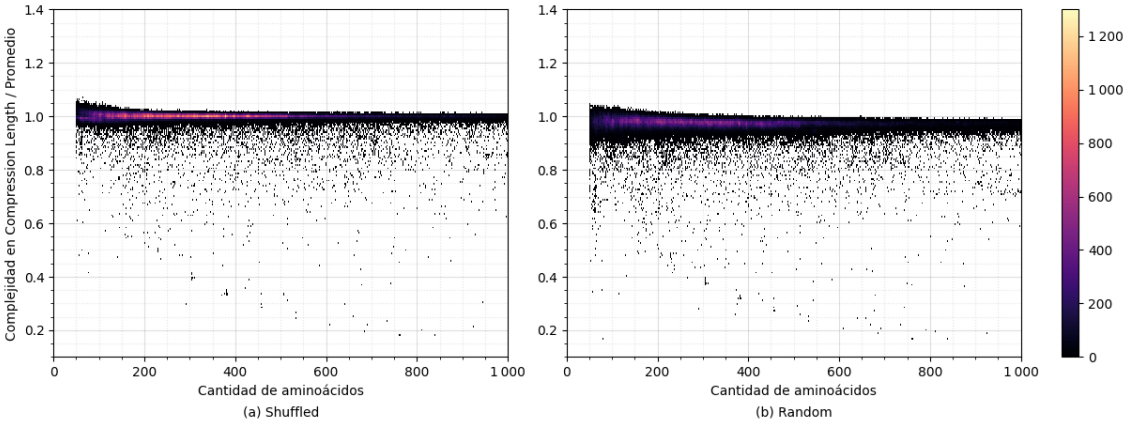

Fig. 3.18: Razón entre la complejidad en Compression length de las secuencias originales y el promedio de complejidad de los casos a) *shuffled* y b) *random*, en función del largo de las secuencias. El color indica la densidad de puntos en cada región con bins de tamaño 2 en el eje x y 0.0065 en el eje y.

En síntesis, los resultados de esta medida 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 gráfico de la razón entre el caso original y el *shuffled*.

4. CONCLUSIONES Y TRABAJO FUTURO

Cumpliendo con los objetivos propuestos, en este trabajo se obtuvieron secuencias de aminoácidos de proteínas naturales de público acceso a partir de UniProt el cual se adaptó para su análisis, generando además conjuntos de datos de control para evaluar diferentes medidas de complejidad. A través del desarrollo de la herramienta *Protein Complexity Study Toolkit*, fue posible automatizar todas las etapas del proceso: desde el filtrado y ordenamiento de las secuencias originales, hasta la generación de casos de testing pseudoaleatorios y de casos de control simples e intuitivos, facilitando tanto el análisis de los datos como la comprensión de los resultados. El sistema de graficación integrado permitió estudiar en profundidad las diferentes medidas de complejidad implementadas, visualizar la diferenciación o la falta de la misma entre conjuntos de datos y evaluar con éxito ocho medidas sobre dos datasets distintos y sus variantes *shuffled*, *random*, *sorted* y *single character*. Asimismo, la herramienta demostró gran flexibilidad para adaptarse a otros conjuntos de datos y para incorporar nuevas medidas de complejidad, consolidándose como un recurso práctico y eficaz para el estudio de la complejidad en secuencias de aminoácidos, independientemente del dataset utilizado o de la medida analizada.

Vimos que, para todas las medidas, el dataset *random* actúa como límite superior para el valor de complejidad, mientras que *single character* funciona como límite inferior (exceptuando la discrepancia, que se comportó de manera inversa). Además, observamos que, a partir de cierto umbral de longitud, para muchas medidas el caso *sorted* también se comporta como límite inferior. Esto nos permite confirmar que las medidas de complejidad se comportan como se espera y nos ayuda a identificar cuáles son más adecuadas para el estudio propuesto. Para el problema que queremos resolver, es preferible elegir una medida que logre distinguir a *sorted* como límite inferior con la menor longitud posible, como es el caso de Compression length, que lo distingue a partir de la longitud 100, o Second order entropy, que lo hace desde la longitud 50; antes que optar por medidas que requieren longitudes mayores (como Bennett, a partir de 300) o que directamente no lo distinguen (como Shannon entropy).

En cuanto a los resultados, vimos que varias medidas nos permitieron diferenciar los casos *random* de los originales, entendiendo que tiene sentido que sean fáciles de diferenciar si se tiene en cuenta que 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 medidas. Aunque dicha superposición mejora en la mayoría de las medidas con el aumento del largo de las secuencias; desapareciendo después del largo 500 en Discrepancia, 400 en Shannon entropy, 500 en Second order entropy y 900 en Compression length.

Sobre los casos *shuffled* no logramos separar satisfactoriamente a estos del caso original, a pesar de algunos outliers en algunas medidas particulares como Ical y Compression length. Esto nos lleva a la conclusión que al menos para las medidas 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* generadas sobre el alfabeto de los aminoácidos, pero que mantienen la misma distribución que las

proteínas funcionales conocidas; aunque sin conservar los patrones que deberían encontrarse en secuencias naturales. A diferencia de las secuencias puramente *random*, cuya distribución es uniforme.

4.1. Trabajo Futuro

- Nuevas Medidas. 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 medida* que diferencie los casos *shuffled* de los originales, la herramienta generada puede utilizarse con nuevas medidas no probadas hasta ahora.

- Discrepancia en bloque. En el presente trabajo, debido al elevado costo temporal práctico asociado al cálculo de la discrepancia en bloque, no se utilizó el conjunto de datos completo `usp_f`, empleado en el resto de las medidas, sino una versión reducida del mismo: `usp_987`. Por este motivo, se proponen las siguientes líneas de trabajo futuro:

  - 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.

- Medidas para una distribución asimétrica. Vimos que muchas veces las medidas 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 medidas. Un posible trabajo futuro sería aplicar nuevas medidas o modificar las existentes para 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 medidas como Icalc y Compression length, el estudio de estos outliers puede servir para entender el trasfondo de lo que está sucediendo y guiarnos mejor a medidas 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 los datos en 5 batches: primero el caso original, después los casos *shuffled*, luego los *random* y finalmente los dos de control. Si bien el contenido de cada batch se computa en paralelo, hasta donde el procesador lo permita; la actual implementación requiere esperar que el batch anterior se compute en su totalidad para comenzar con el siguiente. Una mejora sería que se puedan procesar todos en un solo batch haciendo mejor uso de la totalidad de los recursos del procesador.

- Opinión de expertos. Presentar este trabajo a algún lingüista con el objetivo de obtener su perspectiva sobre el enfoque adoptado. Esto se fundamenta en la posibilidad de interpretar las secuencias de aminoácidos como un lenguaje natural cuyo significado aún no hemos logrado descifrar completamente.

5. ANEXO: DOCUMENTACIÓN DE LA HERRAMIENTA

La herramienta utilizada, *Protein Complexity Study Toolkit*, se encuentra disponible en el siguiente repositorio de GitHub[10].

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 medida 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 de tipo `.fasta` y sus respectivas referencias en largos de secuencia con tipo `.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, Second order entropy y largo en compresión.
- `results` – Carpeta con los resultados de todas las experimentaciones.
- `complexity_metrics.py` – Archivo `Python` con las medidas 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 medida.
- `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_plotting.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 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 medidas Kolmogorov, Bennett, Shannon, Second order entropy 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 en `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 medidas Kolmogorov, Bennett, Shannon, Second order entropy 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 medidas Kolmogorov, Bennett, Shannon, Second order entropy 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 medidas Kolmogorov, Bennett, Shannon, Second order entropy 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 2.3 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] Flissi A, Ricart E, Campart C, Chevalier M, Dufresne Y, Michalik J, Jacques P, Flahaut C, Lisacek F, Leclère V, and Pupin M. Norine: update of the nonribosomal peptide resource. *Nucleic Acids Research*, 48(D1):D465–D469, 2020. PMCID: PMC7145658.

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

[3] 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.

[4] Charles H. Bennett. *Disipation, Information, ComputationalComplexity and the Definiton of Organization*. IBM Research, 1985.

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

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

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

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

[9] Anthony D. Keefe and Jack W. Szostak. Functional proteins from a random-sequence library. *Nature*, 410(6829):715–718, 2001.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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).

[17] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[18] P. Deutsch and J.-L. Gailly. GZIP file format specification version 4.3. RFC 1952, Internet Engineering Task Force, March 1996.