Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Compresores de datos como estimadores de diversidad en repertorios de células T

Tesis de Licenciatura en Ciencias de la Computación

Patricio Tarantino

Director: Esteban Lanzarotti

Buenos Aires, 2019

RESUMEN

La diversidad de receptores de células T presentes en el cuerpo humano es una forma de saber la eficiencia del mismo, ya que nos permite saber a cuántos antígenos distintos es capaz de reconocer. Sin embargo, su medición es difícil pues no todos se presentan en muestras sanguíneas, y más aún, aunque tengamos a los receptores de células T, no es fácil saber qué antígeno reconocerán. Pero sí podemos suponer que receptores de células T con secuencias similares reaccionarán frente a las mismas moléculas externas. Proponemos entonces un modelo basado en los compresores de datos, que son capaces de reconocer patrones en cadenas de texto (cadenas de aminoácidos), que intentarán, dado un conjunto de receptores de células T, estimar su diversidad basado en su composición estructural (sus secuencias de aminoácidos), y a su vez, a cuántos antígenos distintos reconocen. Mejoraremos luego dicho modelo, que llamaremos *CompreScore*, agregando la noción de similitud entre aminoácidos provista por las matrices BLOSUM, que proveerá más información a los compresores a la hora de encontrar patrones y obtener información. Por último, probaremos nuestro modelo *CompreScore* frente a pacientes, y veremos que genera una nueva métrica de diversidad, independiente de las ya conocidas como Shannon o Simpson, y más aún, es también capaz de brindar información en otras dimensiones del paciente, como es su rango etario.

Palabras claves: receptores de células T, sistema inmune, diversidad, compresores de datos, BLOSUM.

ABSTRACT

T-cell repertoire diversity in the human body is a proper way to know how efficient it is, since it allows us to know how many antigens it is able to identify. However, measure this diversity is not an easy task since not all T-cells are present on blood samples, and even wrose, even if we could access all available T-cells, it is not possible to know to which antigen they will react to. But we can assume that T-cell receptors with similar structure will match against the same molecules. We propose, then, a new model based on data compression, which are able to recognize patterns in data strings (amino acids chains). Our model will try, given a set of T-cell receptors, estimate its diversity based on their structural composition, and, at the same time, how many different antigen they can recognize. We will then improve this model, that we will call *CompreScore*, adding the notion of similarity between amino acids, given by BLOSUM matrix, which will provide our compressors with more useful information to find patterns. At last, we will try *CompreScore* against patients' blood samples, and we will see it generates a new diversity metric, independent from the ones we already know such as Shannon and Simpson, Furthermore, our model is also able to give new information from other dimension, such as the patients' age range based on its T-cell diversity.

Keywords: t-cell repertoire diversity, immune system, diversity, data compression, BLOSUM.

AGRADECIMIENTOS

A Esteban, por la oportunidad de este trabajo, por la inmensa paciencia, y por dejarme un aprendizaje que de otra manera hubiese sido imposible adquirir. También a todos los profesores y ayudantes, que durante todos estos años me han moldeado de una forma u otra. A los amigos, que saben bien quiénes son, por las felicidades compartidas y ser un cimiento inamovible en los momentos más difíciles de esta aventura cuando las dudas parecían ganar.
A Papá, Mamá y Le, por la confianza constante, y porque de todo lo que logré y lo que soy, a ellos les corresponde una parte. A Belén, por ser mi mitad, y todo lo demás también.

Índice general

1. INTRODUCCIÓN

Fueron dos los años que pasó Alexander Steven Corbet en la Malasia Británica cazando mariposas. Tenía un doctorado en química inorgánica de la Universidad de Reading (Inglaterra), y se había mudado a Malasia para trabajar en el Instituto de Investigación del Caucho de Malasia, donde oficiaba en la investigación de suelos. Hacia el final de esos dos años durante los cuales perseguía lepidópteros (uno hasta es capaz de imaginar a Corbet con su red) fue que despertó la curiosidad en él y una pregunta en particular: si estuviese dos años más haciéndolo, ¿cuántas nuevas especies lograría descubrir? La cantidad de especies diferentes es lo que llamamos la diversidad del sistema.

Cuando volvió al Reino Unido, con 72 especies de mariposas diferentes, se juntó con el estadístico Ronald Fisher para intentar resolver esta cuestión. Fisher no era ajeno a la biología, y ya había trabajado ampliamente en el campo, haciendo avances en las teorías evolutivas, quizás siendo el más conocido el Principio de Fisher, donde expone por qué la razón entre machos y hembras de las especies conocidas es 1:1. Así publicaron en 1943 "The relation between the number of species and the number of individuals in a random sample of an animal population" [1], el cual introducía por primera vez en la ecología el Unseen species problem, o el Problema de las especies no vistas, el cual trata sobre la estimación del número de especies que habitan en un ecosistema en base a la cantidad de muestras que uno puede extraer. Simplificando a un ejemplo, ¿cuántas especies distintas de animales puede haber en una selva, si durante un mes de observación uno llega a ver sólo 1000 distintas? Diferentes estimadores fueron presentados, y se siguen haciendo hoy en día, pues el problema sigue siendo de importancia. Para nuestros intereses, podemos también considerar el problema de las especies no vistas para otro caso en particular, como es el de la diversidad de receptores de células T.

Las células T (formadas en la médula ósea) son un tipo de glóbulo blanco cruciales para el sistema inmune, ya que trabajan para combatir infecciones virales (entre otros patógenos). Los receptores de estas células son moléculas que se encuentran en su superficie y tienen como función reconocer fragmentos de antígenos (péptidos) ligados a otras moléculas. Una alta diversidad en los receptores de células T significa mayor reconocimiento de péptidos, lo que resulta en un sistema inmune más efectivo (aún con riesgo de provocar enfermedades autoinumunes) y un control más eficiente. No obstante, la medición exacta de la diversidad de receptores de células T (TCR) en la especie humana es imposible con una muestra sanguínea, ya que la diversidad es demasiado alta y la distribución es sesgada.

El conjunto de células T con los mismos receptores define un clonotipo, y aunque en principio éstos podrían definir a antígenos diferentes, se sabe que clonotipos similares en secuencias podrían llegar a unir al mismo antígeno, por lo que para nuestro problema diferentes clonotipos podrían ser tratados como diferentes especies. Sin embargo, considerando que el cuerpo humano tiene alrededor de $4x10^{11}$ células T [2], y que cada clonotipo presente suele aparecer en grandes cantidades, es claro que no es ni siquiera posible estimar un rango de diversidad de TCR en todo el cuerpo humano.

Intentaremos entonces reducir nuestro problema y cambiar el significado de especie para

nuestro ecosistema. Sabemos que los TCR reaccionan ante los péptidos de los antígenos, los cuales son presentados hacia los TCR a través de los complejos mayores de histocompatibilidad (CMH). Así, entonces, diremos que dos TCR pertenecen a la misma especie si reaccionan frente al mismo complejo constituído por el CHM y el péptido del antígeno (y se multiplican en consecuencia). De esta manera, una población de TCR con más especies sería una población más efectiva a la hora de la defensa, pues reconocería más familias de agentes externos.

Sin embargo, el problema es que dadas las cadenas de aminoácidos de los TCR no es fácilmente decodificable qué antígeno reconocen. Nuestro estimador, por lo tanto, intentará, dado un conjunto / muestreo de TCR decidir cuántas especies distintas hay allí, o de manera más exacta, dados dos conjuntos, poder decidir cuál de los dos posee mayor diversidad. A su vez, para mejorar dicho estimador, introduciremos el concepto de matriz de BLOSUM, que define distancia entre aminoácidos en secuencias (basado en la frecuencia conque un aminoácido reemplaza a otro en la propia naturaleza), y con esa información intentaremos ajustar nuestro modelo para que tenga un mejor desempeño.

La posibilidad de poder detectar cambios en la diversidad entre dos conjuntos de células T serviría para hacer un análisis de la respuesta inmunitaria de los procesos de vacunación a los humanos. Este proceso desencadena una respuesta del organismo, la cual forma nuevos anticuerpos que van a actuar neutralizando los agentes infecciosos específicos. Para evaluar su eficacia, se pueden realizar la detección de anticuerpos específicos en el suero del paciente, verificando los que se encontraban presentes semanas antes de la aplicación, y luego semanas después. Como ya dijimos, no se puede acceder a la muestra total de la sangre del paciente, por lo que se hará a través de experimentación *in silico*.

2. REPERTORIO DE CÉLULAS T

Vivimos en un ambiente densamente poblado de agentes, muchos de ellos con la potencialidad de convertirse en agentes patógenos. La función de nuestro sistema inmunitario es defendernos de ellos, y lo logra con creces. Considerando los riesgos a los que estamos expuestos a diario, son pocas las veces que sufrimos procesos infecciosos, y en el caso de que lo hagamos, muchas veces nuestro sistema se encarga de dar una respuesta rápida, gracias a los mecanismos de la inmunidad tanto innata como la adaptativa, que se combinan para hacer frente a estas amenazas [3].

La inmunidad innata es la que comunmente se conoce como las barreras físicas que produce el cuerpo para evitar la entrada de organismos, aunque también son reacciones químicas y biológicas. Los ejemplos más conocidos son la propia piel del ser humano, o los propios tejidos mucosos, y son la primera barrera para evitar la penetración de los microorganismos.

Una de sus características fundamentales es la velocidad. En tan solo las primeras horas o días de iniciado el proceso infeccioso, la inmunidad innata será la encargada de iniciar las actividades para erradicarlo. En caso de no lograr su cometido, intentará contener la infección hasta que entren en acción las defensas del sistema adaptativo (proceso que puede llegar a tardar varios días). Otra de sus características importantes es la no especificidad, ya que defiende de igual manera ante todo tipo de agente externo. Esta es una de las principales diferencias con la inmunidad adaptativa, que es en nuestro caso particular la que más nos interesa.

Son las células T y las células B los protagonistas celulares tanto de nuestro sistema adaptativo como el de todos los mamíferos. Al igual que el resto de las celulas del sistema inmunológico, se originan en la médula ósea a partir de una célula precursora, conocida como célula madre pluripotente hematopoyética (CMPH), o simplemente hemocitoblasto. A partir de este se generan progenitores con un potencial más acotado, como el progenitor mieloide, que dará lugar a células como monocitos, macrófagos y neutrófilos, entre otros, y el progenitor linfoide común (PLC), que es el precursor de los linfocitos B y T (y también de las células *natural killer* (NK)) (Figura 2.1).

Es en los denominados órganos linfáticos primarios donde se da el mayor desarrollo de dichas células. Mientras que los linfocitos B completan este proceso en la médula ósea, las células T lo hacen en el timo, una glándula preparada para brindar un entorno favorable al desarrollo de las células T. Cabe destacar que la función específica del timo fue descubierta recién en 1969 [4]. Es justamente allí donde se produce el reordenamiento de las secuencias de ADN encargadas de la generación de receptores, algo de vital importancia para la diversidad de células T y la eficacia de nuestro sistema inmune (más sobre esto luego).

Nos enfocaremos en los linfocitos T, pues son aquellos que utilizaremos en nuestro estudio. Como ya dijimos, su origen es en la médula ósea, pero su maduración más importante ocurre en el timo. En los seres humanos, esta glándula se ubica en el tórax, y se encuentra totalmente desarrollada al momento de nacer, siendo su momento más productivo en la

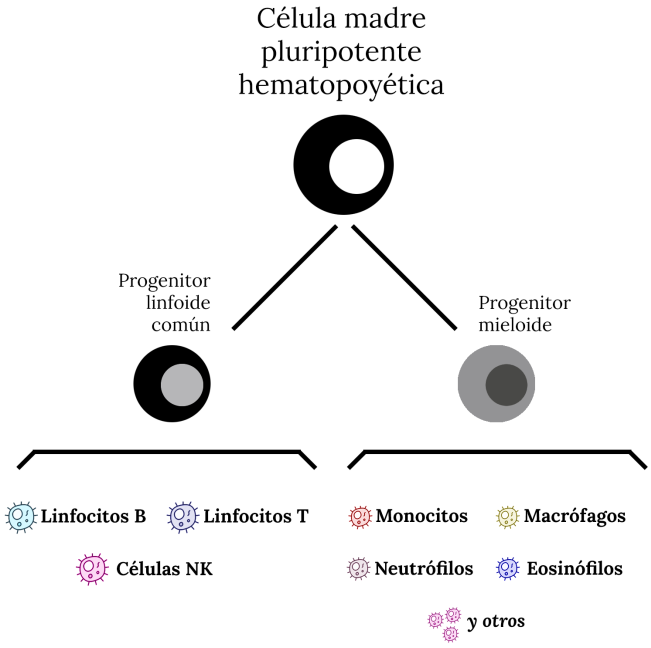

Fig. 2.1: Células progenitoras y sus potenciales.

pubertad. Luego, tanto su producción como su tamaño comienzan a decrecer (es interesante mencionar que dado que el número de células T en el cuerpo humano no decrece significativamente con la edad, se cree que las células T siguen su reproducción en base a las ya maduras que están en el sistema periférico, es decir, aquellas que ya están en el sistema sanguíneo, glándulas linfáticas y amígdalas). El timo no sólo se encarga de la maduración de las células T y el envío de éstas a los órganos secundarios (o periféricos), sino que también aplica un proceso de control para eliminar aquellas células T potencialmente autorreactivas, es decir, aquellas que reaccionen y se activen al reconocer moléculas del propio cuerpo, lo cual sería un error. Desafortunadamente, este objetivo no se cumple en su totalidad, por lo que el sistema inmune empleará otros mecanismos para evitar el accionar de estas células, una vez que estas ya estén en la periferia.

Aunque ya hemos hablado sobre el papel que juegan las células T en el sistema inmune adaptativo e incluso la forma en la que se generan, todavía no sabemos los detalles de su accionar. El proceso en particular que efectúa cada célula T en solitario es complicado pero a la vez muy importante para el reconocimiento de los antígenos que se presentan como amenaza.

Los antígenos (del inglés, *antibody generator*) son moléculas que desencadenan la formación de anticuerpos y pueden causar una respuesta inmune. Son usualmente proteínas, provenientes tanto de bacterias, virus, u otros microorganismos. La porción que nos interesa de estas moléculas, o la que le interesan a las células T, son los epítopos. El epítopo es la parte del antígeno que puede ser reconocida por las células T, y tienen la gran ventaja de identificar de manera única al antígeno del que son parte.

Supongamos, por ejemplo, que tenemos una bacteria A, y alguno de sus epítopos (pues

puede tener varios) está representado por una secuencia lineal de 10 aminoácidos (20^{10} combinaciones posibles). Es muy baja la probabilidad de encontrar esta secuencia en otra proteína de la bacteria A, o en una proteína de otra bacteria B. Y menos aún, en proteínas pertenecientes a virus o parásitos. Por eso decimos que el epítopo permite identificar y marcar casi de manera única a la molécula que representa.

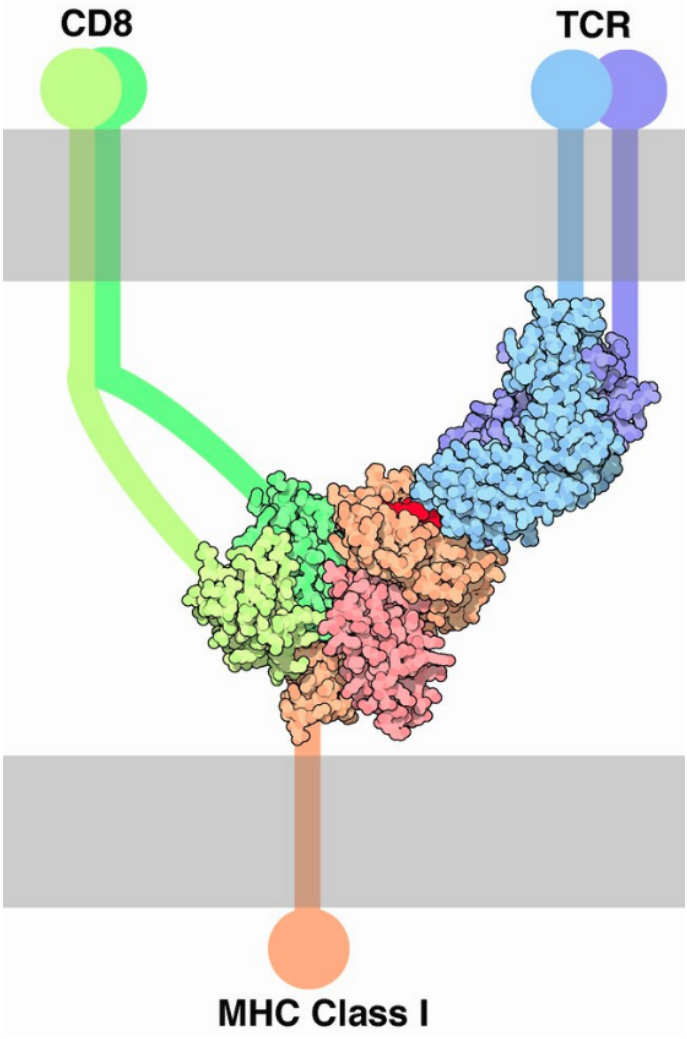

Fig. 2.2: Estructura del TCR con CMH y CD8. Imagen: Wikipedia.

Dada la gran diversidad que hay de dichos epítopos, ¿cómo hace el sistema inmunológico, a través de las células T, para reconocerlos a todos ellos? Ahí entran en juego los millones de clones de linfocitos, que a través de un amplio repertorio serán capaces de reconocer los diferentes antígenos y así activar una respuesta inmune cuando sea necesario.

Para hablar de la manera en que trabajan los linfocitos T y su manera de reconocer estos epítopos, es necesario hablar de la estructura de los receptores de células T (TCR). El TCR está formado por dos cadenas distintas de aminoácidos - las cuales forman un heterodímero - y están ancladas a la membrana celular (figuras 2.2 y 2.3). Existen dos posibles formas de receptores, los TCR$\alpha\beta$, los cuales como su nombre indica están formados por cadenas α y β, y los TCR$\gamma\delta$, formados por cadenas γ y δ. Ya que estos últimos forman una parte

minoritaria de la población total y no trabajamos con ellos, a partir de ahora cuando digamos TCR nos referiremos a los TCR$\alpha\beta$.

La cadena α del TCR tiene aproxidamente 260 aminoácidos, y la β una cantidad más cercana a los 300. La región formada por los primeros 100 a 120 constituyen la región variable, Vα y Vβ, respectivamente (las regiones más próximas a la membrana celular se mantienen iguales en los diferentes clones, por lo que las llamamos regiones constantes, Cα y Cβ). Es en Vα y Vβ donde se encuentra mayor variabilidad y en particular en tres regiones bien definidas, CDR1, CDR2 y CDR3, esta última la que presenta mayor cambio.

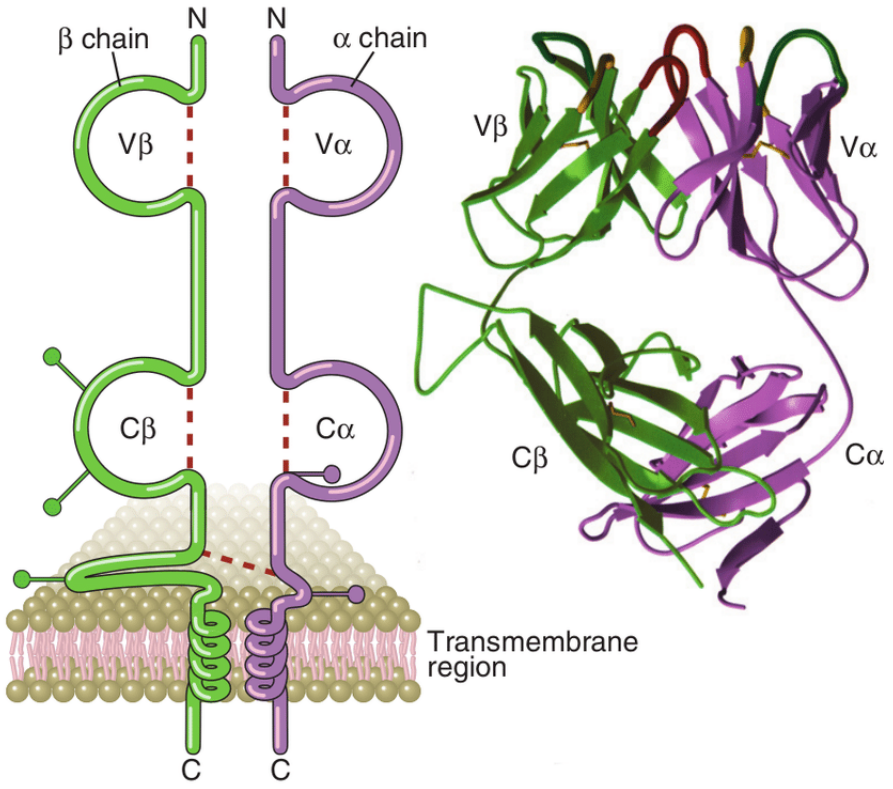

Fig. 2.3: Esquema de los TCR (izquierda) y estructura 3D (derecha). Imagen: Cellular and Molecular Immunology, Abbas, Lichtman, and Pillai, 2012.

Estudios durante los últimos años han logrado mostrar que la región CDR3 será la protagonista al momento del reconocimiento. No obstante, ella sola no será capaz de *ver o entender* al antígeno que está buscando. No solo eso, y aún peor, ni siquiera el receptor de la célula T será capaz de hacerlo. Necesita una ayuda externa para dicha tarea.

Aquí es donde entran en juego las células presentadores de antígenos (CPA), que como su nombre indican, toman un antígeno y se lo presentan a la célula T. Si bien todas las células tienen esta capacidad, algunas de ellas son llamadas *profesionales* por ser ésta su tarea principal, como las células dendríticas. Cuando hablemos de CPA nos referiremos a este último grupo.

Para llevar a cabo esta tarea, las CPA se valen de una molécula proteica que se presenta en la membra celular y que cumple un rol fundamental en todo esto. Esta molécula, llamada

complejo mayor de histocompatibilidad (CMH), es una familia de genes que funcionan como detectores de lo ajeno frente a lo propio. Su descubrimiento se dio en el ámbito de transplantes entre ratones: cuando se daba un transplante de piel entre distintas cepas de ratones, el receptor rechazaba el injerto del dador, desencadenando una respuesta inmune. Lo que provocaba dicha actividad era el CMH, que era la diferencia entre las cepas de ratones. Este fenómeno, conocido como restricción por CMH, les valió el Premio Nobel de Medicina de 1996 a sus dos descubridores, Rolf Zinkernagel y Peter Doherty.

Es así entonces como la CPA capta al antígeno, utilizando el complejo mayor de histocompatibilidad, y presenta al linfocito T el complejo formado por ambos: el CMH y el péptido, que llamaremos pCMH (figura 2.4). Cuando decíamos que la célula T no sería capaz de reconocer al antígeno nos referíamos a esto: necesita de la ayuda del CMH para detectarlo. Como en realidad el linfocito reaccionará frente a esta combinación, decimos que los linfocitos reconocen el complejo formado por ambos, y no sólo al péptido.

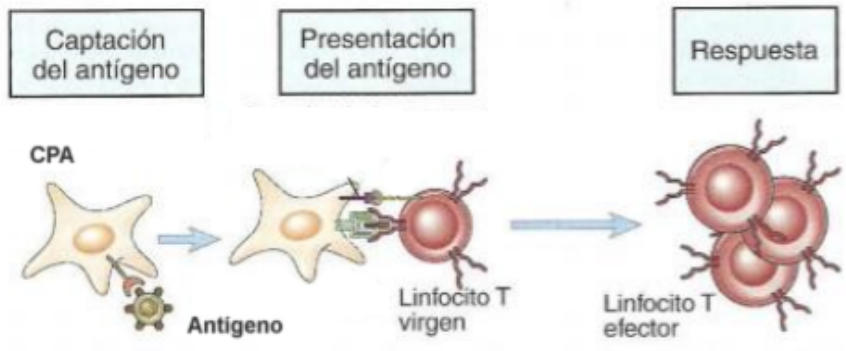

Fig. 2.4: Proceso de la célula presentadora del antígeno.

Ya que los linfocitos T reconocerán solo al par CMH + péptido, la primera restricción para que suceda el reconocimiento es que el CMH debe ser del propio individuo. Esto es una consecuencia del proceso madurativo en el timo: allí, los linfocitos que no son capaces de ver al CMH propio son desechados, para asegurarse que todos aquellos linfocitos que llegan a la madurez sean totalmente útiles. Aunque cada molécula de CMH puede presentar un péptido a la vez (pues su hendidura solo tiene lugar para alojar a uno), cada molécula tiene una especifidad muy amplia, por lo que puede presentar muchos péptidos distintos. Para nuestra experimentación, utilizaremos el CMH humano, llamado HLA (*human leukocit antigen*), y el de los ratones, conocido como H-2 (sistema de histocompatiblidad 2).

Habiendo entendido que para este reconocimiento se necesitan tres partes, nos preguntamos ahora si es posible, sabiendo la estructura del TCR, saber a qué complejo pCMH reconocerá. Más aún, acotando un poco el problema, podemos reducir la estructura del TCR a la región CDR3 del TCRβ, que consideramos la más variable y la más determinante a la hora del reconocimiento.

Por ejemplo, en la siguiente tabla (2.1) vemos que tres regiones de CDR3 reconocen al mismo complejo pCMH, sin embargo, la tercera reconoce y se asocia a uno diferente. A simple vista, no es posible encontrar un patrón en las cadenas de aminoácidos que identifique a qué complejo se asociará.

¿Por qué es importante este problema? Porque un repertorio de TCR de alta diversidad

TCRβ- CDR3	CMH	Péptido
CASSFRSGAETLYF	H-2Db	KAVYNFATC
CASSPDWGDSYEQYF	H-2Db	KAVYNFATC
CASSLDRRNSYNSPLYF	H-2Kb	ASNENMETM
CASSLDRWVYEQYF	H-2Db	KAVYNFATC

Tab. 2.1: Ejemplos de regiones CDR3 de TCRβ y los complejos pCMH que reconoce cada una.

significa un sistema inmune con mayor eficacia. Si el conjunto de todos los clones de TCR que viven en nuestro sistema adaptativo es capaz de reconocer más pCMH, significa que estaremos cubiertos ante una mayor cantidad de antígenos (siempre y cuando no haya fallas en los mecanismos que puedan provocar enfermedades autoinmunes). Sin embargo, medir dicha diversidad no es tarea fácil.

En primer lugar, porque dadas las cadenas α y β de los TCR, no podemos saber a qué pCMH reaccionarán. Suponiendo un caso ideal donde pudiésemos tener el conjunto entero de todos los TCR que forman parte del sistema inmune de un individuo, aún en ese caso imposible, no seríamos capaces de saber contra qué complejos se activarán, y no tendríamos formas de evaluar su efectividad.

Pero más aún, si quisiésemos medir la diversidad de los TCR de un individuo, nos encontraríamos con otro problema. Dado que la diversidad es alta y su distribución no es uniforme, es imposible saber la diversidad real con sólo una muestra sanguínea. La cantidad de células en el cuerpo humano es de alrededor del orden de 10^{12}, con un aproximado de 10^7 clonotipos diferentes. No sólo estamos limitados físicamente por la cantidad de sangre que podemos extraer de un individuo para realizar un análisis, si no también por lo costoso de la secuenciación (lectura) posterior de las células T. Dicho esto, es por eso que los estudios más recientes solo se focalizan en la región CDR3 de TCRβ, ya que se abaratan costos y provee suficiente información para los estimadores.

En los últimos años se han propuesto muchos estimadores diferentes, pero todos tienen el mismo problema de los *unseen species*. Varios estudios han decidido encarar este problema desde el lado de la ecología, al tratarlo como el problema de las *unseen species*.

2.0.1. Diversidad: Las especies no vistas

El problema de las especies no vistas fue planteado por primera vez por Alexander Steven Corbet en la década de 1940. Tenía un doctorado en química inorgánica de la Universidad de Reading (Inglaterra), y se había mudado a Malasia para trabajar en el Instituto de Investigación del Caucho de Malasia, donde oficiaba en la investigación de suelos. Hacia el final de esos dos años durante los cuales perseguía mariposas (uno hasta es capaz de imaginar a Corbet con su red) fue que despertó la curiosidad en él y una pregunta en particular: si estuviese dos años más haciéndolo, ¿cuántas nuevas especies lograría

descubrir? La cantidad de especies diferentes es lo que llamamos la diversidad del sistema.

Cuando volvió al Reino Unido, con 72 especies de mariposas diferentes, se juntó con el estadístico Ronald Fisher para intentar resolver esta cuestión. Fisher no era ajeno a la biología, y ya había trabajado ampliamente en el campo, haciendo avances en las teorías evolutivas, quizás siendo el más conocido el Principio de Fisher, donde expone por qué la razón entre machos y hembras de las especies conocidas es 1:1. Así publicaron en 1943 *"The relation between the number of species and the number of individuals in a random sample of an animal population"*, el cual introducía por primera vez en la ecología el *Unseen species problem* (Problema de las especies no vistas), el cual trata sobre la estimación del número de especies que habitan en un ecosistema en base a la cantidad de muestras que uno puede extraer. Simplificando a un ejemplo, ¿cuántas especies distintas de animales puede haber en una selva, si durante un mes de observación uno llega a ver sólo 1000 distintas? Diferentes estimadores fueron presentados, y se siguen haciendo hoy en día, pues el problema sigue siendo de importancia.

El paralelismo con la diversidad del sistema inmune se puede plantear de la siguiente forma: si en una muestra sanguínea vemos cierta cantidad de clonotipos (ya que estamos secuenciando sólo la región CDR3 de TCRβ, diremos que dos células T son el mismo clonotipo si coinciden sus β-CDR3), ¿cuántos clonotipos habría realmente si pudiésemos duplicar la cantidad de la muestra, o más aún, si tuviésemos acceso al sistema en su totalidad?

Para ello podemos utilizar estimadores, que como su nombre nos indica, dado un muestreo pequeño nos ayudarán a estimar, con cierto grado de precisión, los valores reales de la distribución. Uno de ellos es el Índice de Shannon. Propuesto originalmente en 1948 por Claude Shannon para cuantificar la entropía en telecomunicaciones, aunque posteriormente fue adoptado por la comunidad ecológica para medir diversidad. La idea original indica que a mayor cantidad de letras distintas hay en la cadena, y más parecida es su frecuencia de aparición sobre el largo total, será más difícil predecir cuál será la siguiente en aparecer. La forma de calcularlo es la siguiente,

$$H = -\sum_{i=1}^{R} p_i \ln p_i$$

siendo p_i la frecuencia de aparición de la letra i sobre el total, si habláramos de cadenas de texto. En ecología, p_i sería la cantidad de individuos de la especie i dividido el total de individuos. Para nuestro caso en particular, sería la cantidad de clonotipos de tipo i sobre el total de células secuenciadas. Dado que originalmente el Índice de Shannon mide entropía, la manera de interpretarlo es que a un valor más alto, más entropía hay, y por lo tanto, hay más diversidad (Figura 2.5).

Otro de los índices que tenemos disponibles para medir diversidad es el Índice de Simpson, introducido apenas un año después que el de Shannon por Edward Simpson, esta vez con el fin específico de medir el grado de concentración de individuos cuando éstos son clasificados en especies. El Índice de Simpson no es utilizado sólo en ecología, si no también en economía para medir el tamaño e influencia de corporaciones en relación a la industria y la competencia que hay en ella.

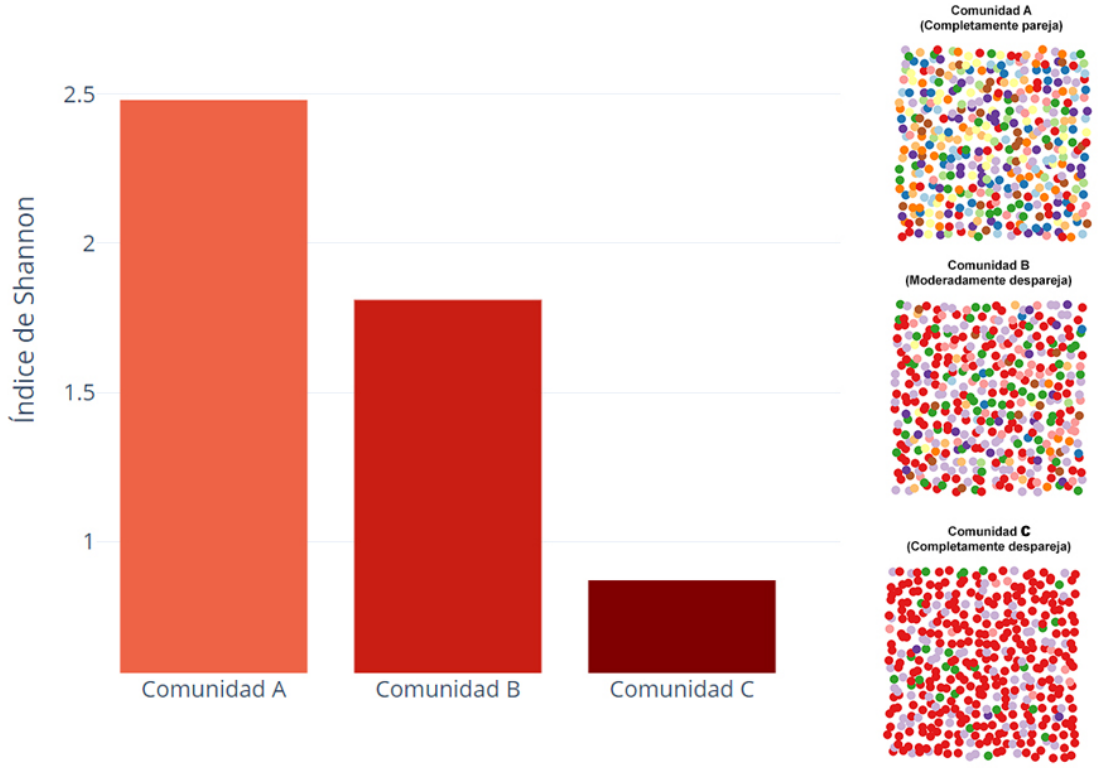

Fig. 2.5: Índice de Shannon para diferentes comunidades, cada una con distintos niveles de diversidad (ordenadas de mayor a menor diversidad).

$$\lambda = \sum_{i=1}^{R} p_i^2$$

Siendo p_i la misma proporción que en el Índice de Shannon, el Índice de Simpson nos da la probabilidad que dos individuos elegidos al azar pertenezcan a la misma especie. El Índice Inverso de Simpson, calculado simplemente como $\frac{1}{\lambda}$, nos da una medida sobre la cantidad de especies.

Como un ejemplo de la forma en la que trabaja Shannon, mostraremos un ejemplo utilizando repertorios de regiones CDR3. Los siguientes conjuntos pertenecen a versiones acotadas de repertorios de pacientes reales.

Como se puede ver en la Tabla 2.2, los conjuntos 1 y 2 tienen una distribución similar si pensamos en la frecuencia con la que aparecen sus CDR3 más populares. En ambos casos, por ejemplo, el CDR3 más popular aparece casi 3.5 veces más que el menos popular. Y la distribución entre ellos es casi lineal. Sin embargo, en el Conjunto 3, el más popular es 50 veces más que el menos, y se pueden ver grandes saltos en su distribución. Ya el segundo baja de 2170 apariciones a solamente 218.

Los índices de Shannon y Simpson nos dan esta pauta. Mientras que en los dos primeros casos los índices son muy parecidos, en el tercer caso cambian considerablemente y nos muestran que el Conjunto 3 tiene menos diversidad, ya que sus valores bajos reflejan menos entropía. Esto puede verse de la siguiente manera: si sacamos del Conjunto 3 un elemento al azar, hay muchísisimas chances de que este pertenezca al CDR3 CASSTRGSWNTGELFF, que es el más popular. Sin embargo, si sacamos uno al azar del Conjunto 1 o del Conjunto 2, será más equitativa la distribución y aunque algunos tendrán mayor probabilidad que otros, será mucho más parejo.

Conjunto 1		Conjunto 2		Conjunto 3	
Ap.	**CDR3**	**Ap.**	**CDR3**	**Ap.**	**CDR3**
92	CASSWGQGAGELFF	226	CASSDRMNTEAFF	2170	CASSTRGSWNTGELFF
75	CASSQFGGAYEQYF	210	CASSSALAGERSYEQYF	218	CASSLEGHQPQHF
61	CASSFGGRTEAFF	185	CASSESSTDTQYF	210	CASSLVGQGFYGYTF
61	CAGTEPGLERTEAFF	155	CASRDSLGTYGYTF	203	CASSIDRWGAVHTEAFF
60	CASSLYGGEQYF	147	CASGLGGDRGKYEQYF	155	CASGTGQFSNQPQHF
59	CASGELGGATEAFF	121	CASSSDSTVHTF	117	CASSYSTDRDTWVNTEAFF
58	CASSLQTSGFYEQYF	109	CASSYVQGVSYEQYF	84	CASSFGYEQYF
48	CATSRGQGRTPLHF	103	CASSQEDGGPTDTQYF	72	CASSLGLAGTDTQYF
42	CASSESSTDTQYF	101	CASSIGAGGPNTGELFF	71	CASRTGTKEFF
38	CASSSALAGERSYEQYF	97	CASSTRDPSSGNTIYF	68	CASSPGQALEKLFF
28	CASSPVVRHGYTF	85	CASSQQAGQAYNSPLHF	66	CASSIKGGTYNEQFF
27	CASGLGGDRGKYEQYF	85	CASSQGGPLNEQFF	56	CASSQDPTDPRLDTQYF
26	CASSDRMNTEAFF	76	CASSRTGAGGNTIYF	45	CASSLSGTSGRANTGELFF
25	CASSTRDPSSGNTIYF	72	CASSYGLGEKLFF	44	CASSLVRTTNTEAFF
25	CASSQEDGGPTDTQYF	66	CATRGTGDEKLFF	43	CASRETSGDPSDEQYF
Índice de Shannon: 2.62		Índice de Shannon: 2.63		Índice de Shannon: 1.66	
Índice de Simpson: 12.84		Índice de Simpson: 12.91		Índice de Simpson: 2.66	

Tab. 2.2: Para cada uno de los tres conjuntos, mostramos cuántas veces aparece repetida (primera columna) cada una de las regiones CDR3 en una muestra de células T. Al final, mostramos el índice de Shannon y el índice de Simpson para cada conjunto de regiones CDR3.

3. COMPRESORES

La función de los compresores, en el ámbito de la computación, es tal como su nombre lo indica, el reducir en volumen una cantidad de información. No nos referimos, sin embargo, a reducir la información en sí misma, si no la forma de representarla (a menudo, expresada en unidades de bits o bytes). Los compresores son, en realidad, algoritmos de ida y vuelta, que permiten no sólo la reducción de la representación de nuestra información, si no el camino inverso también para poder recuperarla y consumirla. Llamaremos *encoder* al proceso de comprimirla y *decoder* al camino inverso, a veces también llamado descomprimir.

Hay que tener en cuenta que los compresores no son agnósticos con el tipo de datos que comprimen (aunque en el fondo son solo bits), y no todos se desempeñan de igual manera. Por un lado, tenemos los compresores sin pérdida (o *lossless*), y por otro, aquellos que sí generan pérdida (pero que dado el tipo de información que manejamos esta pérdida está permitida). En este último caso solemos encontrarnos con compresores de material multimedia, como imágenes, audio o video. Esta limitación es también impuesta por la misma naturaleza de la información, ya que estos tipos de datos son en realidad señales analógicas continuas que debemos codificar de alguna u otra manera para poder almacenarla de forma discreta (con finitos datos).

Sin embargo, en el caso que sí nos interesa, que es el de cadenas de texto, sí podemos utilizar compresores *lossless* (o exactos). ¿Pero qué significa exactamente esto? Definimos a la función

$$f(x_1, x_2, ..., x_n) = y_1, y_2, ..., y_m$$

como nuestro compresor. Recibe una lista finita de n símbolos y devuelve una lista de m símbolos. Nuestra f cumple a su vez dos condiciones. La primera es que $m \leq n$, ya que esto nos valida que f comprime y devuelve una representación de la información en una cantidad reducida de datos. La segunda es que f sea biyectiva, lo que nos asegura que es sin pérdida. Esto nos garantiza que dado un $y_1, ..., y_m$, podemos saber qué información original $x_1, ..., x_n$ está almacenando, y hacer el camino inverso para poder obtenerla.

La forma en la que definimos nuestra f es genérica y no pertenece a ningún compresor en particular. De hecho, cualquier función que sepamos que cumpla estas dos condiciones la podremos considerar una función compresor. En general los compresores utilizan dos vías distintas para lograr esto, y podemos dividirlos en base a cuál usan. El *encoding* por diccionario o el *encoding* estadístico.

El *encoding* por diccionario es muy sencillo: el algoritmo busca cadenas de símbolos, que ya están predefinidas en un diccionario, en la cadena *input*. Si encuentra dichas cadenas, las sustituye por la referencia en el diccionario, la cual debe ser siempre más chica que la cadena en sí. Para realizar la decompresión, se realiza lo mismo en el sentido inverso.

Supongamos, por ejemplo, que queremos comprimir cadenas de texto las cuales sabemos de antemano que contienen varios espacios en blanco seguidos. La clave de esto es que

sabemos esta característica de antemano, lo que nos permite pre-construir el diccionario. Podríamos sugerir el siguiente diccionario:

Clave	Cadena
1	␣
2	␣ ␣
3	␣ ␣ ␣

Tab. 3.1: Diccionario para reemplazar espacios en blanco.

Por lo tanto, al utilizar este diccionario para comprimir las siguientes cadenas, veríamos estos resultados

Input	Output	Long. input	Long. output
El ␣ ␣ auto ␣ ␣ rojo	El2auto2rojo	14	12
El ␣ ␣ ␣ ␣ auto ␣ ␣ ␣ chocó	El22auto3chocó	17	14
Auto	Auto	4	4

Tab. 3.2: Input y output de compresor por diccionario.

Como se aprecia, el ejemplo no es muy convincente. Ni siquiera comprime en todos los casos, como se ve en la tercera fila. Estos compresores requieren tener mucha información previa de los inputs a utilizar, porque es vital la construcción de un diccionario que logre su cometido, ya que ahí es donde reside la eficacia del compresor. Podríamos agregar una nueva clave a nuestro diccionario, por ejemplo, la clave A para representar *auto*, sin embargo, deberíamos asegurarnos que dicha clave no aparezca en ninguno de nuestros inputs, porque si no, no sabríamos la función que cumpliría ese símbolo (podría ser dato o clave).

Los compresores estadísticos (o adaptativos) descubren ellos mismos los patrones y la redundancia en el input. Una de las formas en las que trabajan es que a cada símbolo (o cadena de símbolos) se le asigna un código basado en la probabilidad de que dicho símbolo aparezca, asignádosele códigos más cortos a los símbolos con mayor frecuencia.

Supongamos por ejemplo que queremos comprimir textos escritos en el idioma castellano (para simplificar el ejemplo, aceptaremos sólo cadenas de texto en castellano en letras minúsculas y con espacio, sin símbolos de puntuación). Un input podría ser *el este y el oeste* y otro podría ser *el norte y el sur*. Considerando que en la codificación ASCII cada letra (y el espacio en blanco) necesita 1 byte para ser almaceada (8 bits), la primera cadena necesitaría 18 bytes (144 bits), mientras que la segunda necesitaría 17 bytes (136 bits). No obstante, ¿por qué utilizar la codificación ASCII, que le da la misma importancia a cada uno de los símbolos, utilizando 8 bits para almacenarlos?

Sabiendo que en el idioma español la frecuencia de aparición de ciertas letras es mayor, ¿no podríamos aprovecharnos de eso? Así es como podemos sugerir la creación de un compresor de manera intuitiva: podríamos asignar códigos más cortos a la vocales *a* y *e*, y a las consonantes *s* y *r*, que son de las cuatro letras con mayor frecuencia de aparición. La manera de realizar esto es con prefijos. Podríamos entonces sugerir la siguiente codificación:

- a $\rightarrow$ 000
- e $\rightarrow$ 001
- s $\rightarrow$ 010
- r $\rightarrow$ 011

mientras que el resto de las letras tendrá una codificación de tipo 111XXXXX, es decir, con un prefijo de 111, que indicará al momento de la lectura que no es ninguno de los símbolos más frecuentes, y que debemos leer los siguientes 5 símbolos para decodificar dicho dato. Ahora, utilizando esta codificación, podemos ver que las cadenas anteriores necesitan menos cantidad de bits para ser almacenadas. Mientras que *el este y el oeste* antes necesitaba 144 bits, ahora basta con 104 bits (una reducción del 30 %), y *el norte y el sur* ahora necesita 106 bits contra los 136 originales (una reducción del 22 %).

Si bien nosotros hemos construído estos prefijos de manera intuitiva y poco exacta, se conocen algoritmos para hacerlo de forma óptima, el más popular siendo la Codificación de Huffman. El algoritmo propuesto por Huffman es el más eficiente para la creación de códigos prefijos: no hay otra forma alternativa de representar los símbolos de manera más pequeña considerando su frecuencia de aparición (más aún, si la entrada al algoritmo de Huffman está ordenada, su complejidad es lineal, lo que lo hace muy práctico).

Esto nos da un panorama muy genérico de la forma en la que funcionan los algoritmos de compresión, sin embargo, los que son usados en la práctica son más complejos, ya que tienen en cuenta otras propiedades de los símbolos como el contexto en el que aparecen, entre otras cosas. Para nuestra experimentación utilizamos cuatro algoritmos de compresión que están implementados como bibliotecas de Python, lo que facilita su uso. Los considerados fueron **ZLIB**, **LZMA**, **GZIP** y **BZIP**.

3.0.1. ZLIB

El algoritmo de compresión ZLIB fue originalmente escrito por Jean-loup Gailly y Mark Adler en el lenguaje C, y lanzado al público por primera vez en mayo de 1995. Ambos trabajarían luego en GZIP (como vamos a ver, GZIP usa ZLIB). Al día de hoy, ZLIB sigue siendo mantenido por Mark Adler, quien sigue colaborando activamente en el proyecto.

ZLIB es una implementación del algoritmo Deflate, creado por Phil Katz para su herramienta PZKIP (circa 1993). Este algoritmo hace una doble codificación, primero utilizando el algoritmo LZ77 y luego la codificación de Huffman previamente mencionada (figura 3.1).

La idea principal de LZ77 (original del dúo israleí Lempel-Ziv) es la de buscar cadenas de texto repetidas y reemplazarlas por punteros a la ubicación original. La manera de

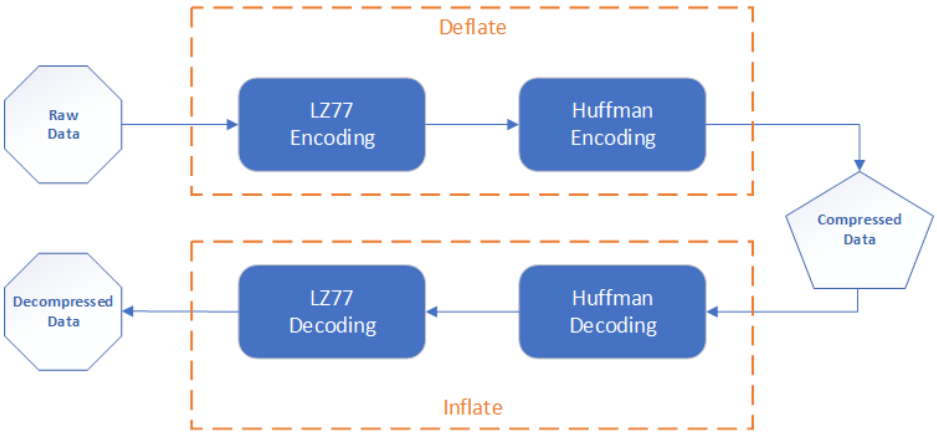

Fig. 3.1: Panorama general de ZLIB y el algoritmo deflate. Imagen: Understanding zlib (www.euccas.me)

hacer esto es con una ventana deslizante, teniendo un *search buffer* y un *look-ahead buffer*, ambos de tamaño limitado. En el *search buffer*, el algoritmo irá almacenando los datos más recientes que fueron comprimidos, mientras que en el *look-ahead buffer* tenemos los próximos a comprimir. El tamaño de ambos *buffers* determinan el nivel de compresión y desempeño del algoritmo: si son muy pequeños, las pasadas serán más rápidas pero el algoritmo verá "menos" partes del *input*, por lo que encontrará menos cadenas repetidas. En caso contrario, tendrá una visión más global (a mayor buffer), pero será más lenta su ejecución. En particular en Python, ZLIB está implementado por defecto con un *buffer* de 4kb.

Cada vez que el algoritmo encuentre una cadena en el *look-ahead buffer* que está presente en el *search buffer*, la reemplazará con un par *length-distance*. Dicho par tendrá dos valores: la distancia desde ese punto (en símbolos) a la aparición de la cadena original, y el largo de la misma.

Usemos de ejemplo el estribillo de la canción *Yellow submarine*, de *The Beatles*:

```
0: We all live in a yellow submarine
33: Yellow submarine, yellow submarine
68: We all live in a yellow submarine
102: Yellow submarine, yellow submarine
```

(Hemos agregado al principio de cada línea la posición de los símbolos para facilitar la lectura).

Luego de ejecutar el algoritmo LZ77, este será nuestro *output*:

```
We all live in a yellow submarine
[17,16], [18,16]
```

```
[68,17] [34,16]
[17,16], [18,16]
```

Como se ve, la cantidad de símbolos necesarios para almacenar dicho fragmento bajó considerablemente. Sin embargo, ZLIB no termina aquí, si no que aplica una codificación de Huffman para reducir aún más, basándose en la frecuencia de aparición de los símbolos para representarlos de una forma eficiente.

3.0.2. LZMA

Aunque en su nombre figura (otra vez) el dúo israelí Abraham Lempel y Jacob Ziv (LZMA viene de *Lempel-Ziv-Markov chain Algorithm*) fue creado por Igor Pavlov y fue utilizado en el popular software *7-Zip*.

La forma de trabajar de LZMA es muy similar a ZLIB. En primer lugar, funciona con una variante de LZ77. Los pasos son similares, con la diferencia que en este caso los tamaños de los buffers son mucho más amplios. En segundo lugar, no utiliza la Codificación de Huffman, si no Rango de Codificación, una variable de la Codificación Aritmética que funciona con números enteros en vez de racionales entre 0 y 1.

La Codificación por Rango funciona de la siguiente manera para un *input* de X símbolos de largo: se toma un segmento de dos números enteros, 0 y N. Se subdivide en M segmentos, si es que tenemos M símbolos distintos, cada uno proporcional a la probabilidad de que aparezca cada símbolo. Para codificar, tomamos el primer símbolo M_1 y elegimos su segmento. Volvemos a dividir dicho segmento S_1 como hicimos con el rango inicial (en base a las probabilidades), y elegimos el correspondiente al segundo símbolo S_2, con $S_2 \subset S_1$. Repetimos el proceso sucesivamente hasta llegar a S_X, siendo este el último segmento, el cual cumplirá que $S_X \subset S_{X-1} \subset ... \subset S_2 \subset S_1$. Teniendo S_X, elegiremos un $n \in S_X$ tal que la codificación en bits de n sea la menor posible. Se puede ver un ejemplo con la cadena AABA[EOM] en la Figura 3.2, con las siguientes probabilidades: A: 0.6; B: 0.2, [EOM]: 0.2.

La forma de calcular las probabilidades para los rangos es utilizando Cadenas de Markov, de ahí la inclusión en su nombre. Como la probabilidad cambia dependiendo del símbolo anterior, decimos que LZMA es *context aware*, ya que importa el contexto en el que se hace presente cada símbolo. A su vez, esto provoca que haya varias posibles formas de codificar cada cadena de símbolos, por lo que se intenta implementar con programación dinámica para poder elegir la manera óptima.

3.0.3. GZIP

Nuestro tercer algoritmo de compresión es GZIP. Como ya mencionamos, comparte los mismos autores que ZLIB, y en realidad lo utiliza internamente. En pocas palabras, GZIP es otra implementación de algoritmo Deflate.

La diferencia de GZIP con ZLIB no es en su algoritmo, si no que utiliza un *wrapper* diferente para almacenar la metadata necesaria para luego decomprimir los datos. Aunque

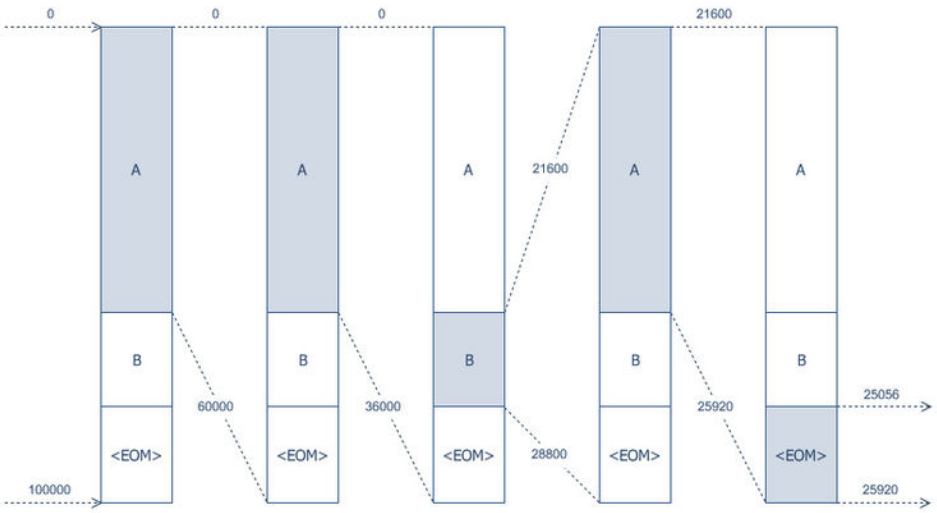

Fig. 3.2: Ejemplo de Codificación por Rango. Cada rango está incluído en el rango del símbolo anterior. Imagen: Wikipedia

esta diferencia parezca que puede ser ignorada, como veremos luego será importante para nuestro uso en particular.

Como bien dice la documentación de ZLIB, GZIP es una implementación para la compresión de archivos, y entre su metadata guarda información de los mismos (como última modificación, por mencionar alguna), mientras que ZLIB no está optimizada para ningún tipo de datos en particular.

3.0.4. BZIP

Debido a como funciona, BZIP es el más distinto a los otros tres que ya vimos. La implementación de BZIP usa la Compresión de Burrows-Wheeler como base. Publicada por David Wheeler y Michael Burrows en 1994, esta compresión hace un fuerte uso de la permutación de los símbolos para agruparlos de manera que se pueda sacar mayor provecho al momento de comprimirlos bajo algún tipo de codificación.

La forma en la que funciona la Compresión de Burrows-Wheeler es sencilla: primero se generan todas las posibles rotaciones del texto a comprimir. Una vez que se tienen todas las rotaciones, se las ordena de manera lexicográfica, y por último, nos quedamos con el último caracter de estas rotaciones en el orden que obtuvimos. Veamos el siguiente ejemplo con la palabra *amamantar* (el carácter ^ indica el comienzo de línea, y | indica fin de línea). La lista con todas las rotaciones posibles sería la siguiente:

```
^amamantar|
|^amamantar
r|^amamanta
ar|^amamant
```

```
tar|^amaman
ntar|^amama
antar|^amam
mantar|^ama
amantar|^am
mamantar|^a
amamantar|^
```

Si las ordenamos lexicográficamente, nos queda la siguiente lista:

```
^amamantar|
amamantar|^
amantar|^am
antar|^amam
ar|^amamant
mamantar|^a
mantar|^ama
ntar|^amama
r|^amamanta
tar|^amaman
|^amamantar
```

y si tomamos los últimos símbolos de cada uno de estas cadenas, nos queda la cadena: $|\hat{}mmtaaaanr$. Como vimos al principio del capítulo, este *output* intermedio está optimizado para usar compresores por diccionario. Por ejemplo, usando *run-length encoding* podríamos comprimir esta cadena a $|\hat{}2mt4anr$. Ahora tenemos que hacer el proceso inverso, de manera cuidadosa, para recuperar la cadena original. La manera de descomprimir sería ir símbolo por símbolo, y si es un número, repetir el siguiente símbolo esa cantidad de veces. En cambio, si el símbolo anterior es un símbolo no-numeral, y el siguiente también, entonces el siguiente solo se repite una sola vez. En nuestro ejemplo, vemos el 2, y repetimos la m dos veces, luego la t una sola vez, luego la a cuatro veces, y la n y la r una vez cada una. En nuestro ejemplo, $2m$ se transformaría en mm, luego la t se agrega (no hay número precedente), quedando mmt. Lo siguiente, $4a$, nos hace agregar 4 veces la letra a, quedando $mmtaaaa$, y finalmente, agregamos los últimos dos símbolos sueltos, n y r, recuperando nuestra cadena original $mmtaaaanr$.

En particular, la implementación de BZIP, una vez llegada a esta etapa, utiliza Codificación de Huffman para realizar la compresión final.

3.0.5. BLOSUM

BLOSUM no es un compresor ni mucho menos, pero lo utilizaremos para facilitar la tarea de estos últimos a la hora de buscar patrones entre cadenas de aminoácidos. BLOSUM (de **BLO**cks **SU**bstitution **M**atrix) es una matriz de sustitución para el alineamiento de secuencias de proteínas.

Por un lado, el alineamiento de secuencias es una forma de mostrar y comparar dos secuencias distintas, y poder representar estas similitudes de una manera visual (que a su vez, nos permitirá asignarle un puntaje a esa similitud, concepto que nos servirá luego para comparar las cadenas de manera numérica). En el siguiente ejemplo, con las secuencias GEKLYECNCERSKAFASC y YECNCHKAFAFH, el alineamiento quedaría de la forma en la que se ve en la Figura 3.3.

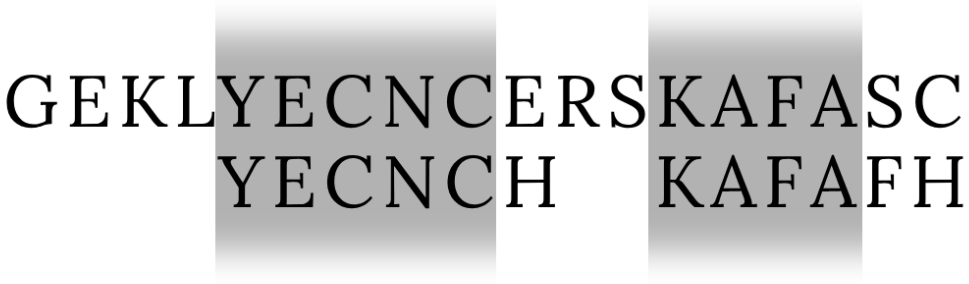

Fig. 3.3: Alineamiento de secuencias para GEKLYECNCERSKAFASC y YECNCHKAFAFH.

El alineamiento permite la inserción de espacios en lugares estratégicos para que las zonas con la misma estructura se alineen perfectamente (o en zonas parecidas, como veremos luego al realizar ciertos reemplazos basados en similitudes químicas de los aminoácidos). Los algoritmos para realizar este alineamiento son variados y debido a su complejidad, se usan diferentes alternativas que van desde la programación dinámica hasta métodos probabílisticos y diferentes heurísticas. Sin embargo, una vez finalizado, nos da un panorama de la similitud estructural de las dos secuencias, y a su vez, de su similitud funcional, como por ejemplo, la especificidad de unión de los receptores de células T (que es lo que nos interesa).

La construcción de las matrices BLOSUM se basan en alineamientos ya calculados en la base de datos BLOCKS, y utilizan la frecuencia de cambio de un aminoácido a otro (en dichos alineamientos) como la probabilidad de una mutación natural (por proceso evolutivo) de estos aminoácidos. La forma en la que se calcula cada valor en cada posición de la matriz es la siguiente (notar que es una matriz simétrica, pues BLOSUM^t = BLOSUM).

$$a_{ij} = \left(\frac{1}{\lambda}\right) \log\left(\frac{p_{ij}}{q_i * q_j}\right)$$

Cada fila y columna representarán un aminoácido, por eso nuestra matriz tendrá 20 filas y columnas. En particular, a_{ij} es la relación que hay entre ellos y se calcula de la siguiente manera: p_{ij} es la probabilidad en que i y j se reemplacen el uno al otro. Esta probabilidad está basada en la frecuencia de cambio de aminoácidos en cada una de las secuencias observadas en la base de datos BLOCKS, y es la cantidad de veces que un aminoácido cambia (en el proceso evolutivo) por otro de manera natural, dividido el total de apariciones de dicho aminoácido. Por otro lado, q_i y q_j son simplementes la probabilidad de encontrar a cada uno de estos aminoácidos en cualquiera secuencia de manera aleatoria (ya que hay

aminoácidos que aparecen más seguidos que otros, y esto es una manera de *normalizar* esa disparidad). Por último, $\frac{1}{\lambda}$ es simplemente un factor de redondeo.

Diferentes versiones de matrices BLOSUM existen, y difieren en la base de datos que utilizan. Nosotros utilizaremos BLOSUM62.

$$
\begin{bmatrix}
 & A & R & N & D & C & Q & E & G & H & I & L & K & M & F & P & S & T & W & Y & V \\
A & 4 & -1 & -2 & -2 & 0 & -1 & -1 & 0 & -2 & -1 & -1 & -1 & -1 & -2 & -1 & 1 & 0 & -3 & -2 & 0 \\
R & -1 & 5 & 0 & -2 & -3 & 1 & 0 & -2 & 0 & -3 & -2 & 2 & -1 & -3 & -2 & -1 & -1 & -3 & -2 & -3 \\
N & -2 & 0 & 6 & 1 & -3 & 0 & 0 & 0 & 1 & -3 & -3 & 0 & -2 & -3 & -2 & 1 & 0 & -4 & -2 & -3 \\
D & -2 & -2 & 1 & 6 & -3 & 0 & 2 & -1 & -1 & -3 & -4 & -1 & -3 & -3 & -1 & 0 & -1 & -4 & -3 & -3 \\
C & 0 & -3 & -3 & -3 & 9 & -3 & -4 & -3 & -3 & -1 & -1 & -3 & -1 & -2 & -3 & -1 & -1 & -2 & -2 & -1 \\
Q & -1 & 1 & 0 & 0 & -3 & 5 & 2 & -2 & 0 & -3 & -2 & 1 & 0 & -3 & -1 & 0 & -1 & -2 & -1 & -2 \\
E & -1 & 0 & 0 & 2 & -4 & 2 & 5 & -2 & 0 & -3 & -3 & 1 & -2 & -3 & -1 & 0 & -1 & -3 & -2 & -2 \\
G & 0 & -2 & 0 & -1 & -3 & -2 & -2 & 6 & -2 & -4 & -4 & -2 & -3 & -3 & -2 & 0 & -2 & -2 & -3 & -3 \\
H & -2 & 0 & 1 & -1 & -3 & 0 & 0 & -2 & 8 & -3 & -3 & -1 & -2 & -1 & -2 & -1 & -2 & -2 & 2 & -3 \\
I & -1 & -3 & -3 & -3 & -1 & -3 & -3 & -4 & -3 & 4 & 2 & -3 & 1 & 0 & -3 & -2 & -1 & -3 & -1 & 3 \\
L & -1 & -2 & -3 & -4 & -1 & -2 & -3 & -4 & -3 & 2 & 4 & -2 & 2 & 0 & -3 & -2 & -1 & -2 & -1 & 1 \\
K & -1 & 2 & 0 & -1 & -3 & 1 & 1 & -2 & -1 & -3 & -2 & 5 & -1 & -3 & -1 & 0 & -1 & -3 & -2 & -2 \\
M & -1 & -1 & -2 & -3 & -1 & 0 & -2 & -3 & -2 & 1 & 2 & -1 & 5 & 0 & -2 & -1 & -1 & -1 & -1 & 1 \\
F & -2 & -3 & -3 & -3 & -2 & -3 & -3 & -3 & -1 & 0 & 0 & -3 & 0 & 6 & -4 & -2 & -2 & 1 & 3 & -1 \\
P & -1 & -2 & -2 & -1 & -3 & -1 & -1 & -2 & -2 & -3 & -3 & -1 & -2 & -4 & 7 & -1 & -1 & -4 & -3 & -2 \\
S & 1 & -1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & -2 & -2 & 0 & -1 & -2 & -1 & 4 & 1 & -3 & -2 & -2 \\
T & 0 & -1 & 0 & -1 & -1 & -1 & -1 & -2 & -2 & -1 & -1 & -1 & -1 & -2 & -1 & 1 & 5 & -2 & -2 & 0 \\
W & -3 & -3 & -4 & -4 & -2 & -2 & -3 & -2 & -2 & -3 & -2 & -3 & -1 & 1 & -4 & -3 & -2 & 11 & 2 & -3 \\
Y & -2 & -2 & -2 & -3 & -2 & -1 & -2 & -3 & 2 & -1 & -1 & -2 & -1 & 3 & -3 & -2 & -2 & 2 & 7 & -1 \\
V & 0 & -3 & -3 & -3 & -1 & -2 & -2 & -3 & -3 & 3 & 1 & -2 & 1 & -1 & -2 & -2 & 0 & -3 & -1 & 4
\end{bmatrix}
\quad BLOSUM62 (3.1)
$$

La forma en la que nosotros utilizaremos BLOSUM62 será para crear una matriz de distancia entre cada uno de los aminoácidos. Notemos que los a_{ij} de la matriz no son distancias y no podríamos usarlos como tal, ya que no cumplen las propiedades de una distancia. Esto se ve claramente en los a_{ij} en donde $i = j$. En esos casos, debería ser 0 (por la propiedad de $d(x, x) = 0$). Sin embargo, la diagonal de la matrix BLOSUM62 no es nula.

Para calcular la distancia entre dos aminoácidos, calcularemos la norma vectorial de su resta, tomando como el vector de cada aminoácido su fila correspondiente en BLOSUM62.

De esta manera, nos aseguramos que la distancia entre el mismo aminoácido de siempre 0 (pues la resta de sus vectores dará $\vec{0}$). Sea B = BLOSUM, y D nuestra matriz de distancias, ésta quedará formada de la siguiente forma:

$$D_{i,j} = ||B_i - B_j||$$

Queda claro que $D_{i,j} = 0$ si $i = j$, que serán los elementos de la diagonal. Además, como ejemplo, la distancia entre A y R será

$$\|A - R\| = 10{,}82$$

y estará ubicada en $D_{1,2}$. La matriz con estas distancias calculadas queda de la siguiente manera:

	A	*R*	*N*	*D*	*C*	*Q*	*E*	*G*	*H*	*I*	*L*	*K*	*M*	*F*	*P*	*S*	*T*	*W*	*Y*	*V*
A	0,00	10,82	12,33	12,73	11,87	10,86	11,62	10,49	14,39	11,27	10,91	10,44	10,15	14,11	11,31	6,56	7,21	18,49	14,76	9,38
R	10,82	0,00	9,43	11,96	17,26	6,86	8,60	12,69	11,22	16,06	14,56	4,90	12,73	16,55	12,61	9,59	10,82	19,31	15,46	14,93
N	12,33	9,43	0,00	8,37	18,36	9,38	8,89	10,86	10,72	17,64	17,18	9,33	15,46	17,92	13,34	7,81	10,68	21,31	16,85	16,61
D	12,73	11,96	8,37	0,00	18,30	10,00	6,86	11,49	13,30	17,69	17,86	10,82	16,52	18,30	11,83	9,43	11,83	21,35	18,00	16,61
C	11,87	17,26	18,36	18,30	0,00	17,86	19,03	16,64	18,97	12,81	12,65	17,49	13,56	15,30	17,00	14,63	13,45	18,57	16,88	12,69
Q	10,86	6,86	9,38	10,00	17,86	0,00	5,57	13,19	11,09	16,28	15,00	6,24	12,21	16,64	12,08	8,54	10,58	19,03	14,90	14,70
E	11,62	8,60	8,89	6,86	19,03	5,57	0,00	12,85	11,58	17,20	16,67	7,07	14,70	17,72	11,79	8,83	11,18	20,47	16,46	15,65
G	10,49	12,69	10,86	11,49	16,64	13,19	12,85	0,00	14,66	17,18	16,88	12,77	16,03	16,82	12,81	9,95	12,49	18,28	17,15	16,00
H	14,39	11,22	10,72	13,30	18,97	11,09	11,58	14,66	0,00	17,72	17,09	12,33	15,56	14,97	15,39	12,57	14,18	18,57	11,53	17,12
I	11,27	16,06	17,64	17,69	12,81	16,28	17,20	17,18	17,72	0,00	4,24	16,06	7,48	10,68	16,03	14,49	11,70	18,08	13,75	3,32
L	10,91	14,56	17,18	17,86	12,65	15,00	16,67	16,88	17,09	4,24	0,00	14,76	5,29	10,10	15,78	13,93	11,36	16,76	13,00	5,39
K	10,44	4,90	9,33	10,82	17,49	6,24	7,07	12,77	12,33	16,06	14,76	0,00	12,73	16,97	11,62	8,49	10,34	19,82	15,97	14,53
M	10,15	12,73	15,46	16,52	13,56	12,21	14,70	16,03	15,56	7,48	5,29	12,73	0,00	10,86	14,87	12,08	10,44	16,46	12,61	7,00
F	14,11	16,55	17,92	18,30	15,30	16,64	17,72	16,82	14,97	10,68	10,10	16,97	10,86	0,00	18,14	15,62	14,32	12,29	6,86	11,70
P	11,31	12,61	13,34	11,83	17,00	12,08	11,79	12,81	15,39	16,03	15,78	11,62	14,87	18,14	0,00	11,36	11,66	20,78	17,94	14,49
S	6,56	9,59	7,81	9,43	14,63	8,54	8,83	9,95	12,57	14,49	13,93	8,49	12,08	15,62	11,36	0,00	6,56	19,47	15,46	13,00
T	7,21	10,82	10,68	11,83	13,45	10,58	11,18	12,49	14,18	11,70	11,36	10,34	10,44	14,32	11,66	6,56	0,00	18,17	14,76	10,00
W	18,49	19,31	21,31	21,35	18,57	19,03	20,47	18,28	18,57	18,08	16,76	19,82	16,46	12,29	20,78	19,47	18,17	0,00	12,17	18,22
Y	14,76	15,46	16,85	18,00	16,88	14,90	16,46	17,15	11,53	13,75	13,00	15,97	12,61	6,86	17,94	15,46	14,76	12,17	0,00	13,86
V	9,38	14,93	16,61	16,61	12,69	14,70	15,65	16,00	17,12	3,32	5,39	14,53	7,00	11,70	14,49	13,00	10,00	18,22	13,86	0,00

(3.2)

Para comprender más rápido la información que contiene esta matriz, podemos graficarla en un grafo en donde cada nodo es un aminoácido, intentando mantener la distancia real que informa la matriz (Figura 3.4). Como se ve, los más cercanos a A son S, R y T, mientras que W suele ser el más alejado de todos los demás.

Esto nos da un panorama claro y nos permite pensar en formas de agrupar los diferentes aminoácidos en *clusters* basados en su cercanía. ¿Pero por qué quisiéramos hacer esto y qué relación tendría con los compresores, que son el tema central de este capítulo?

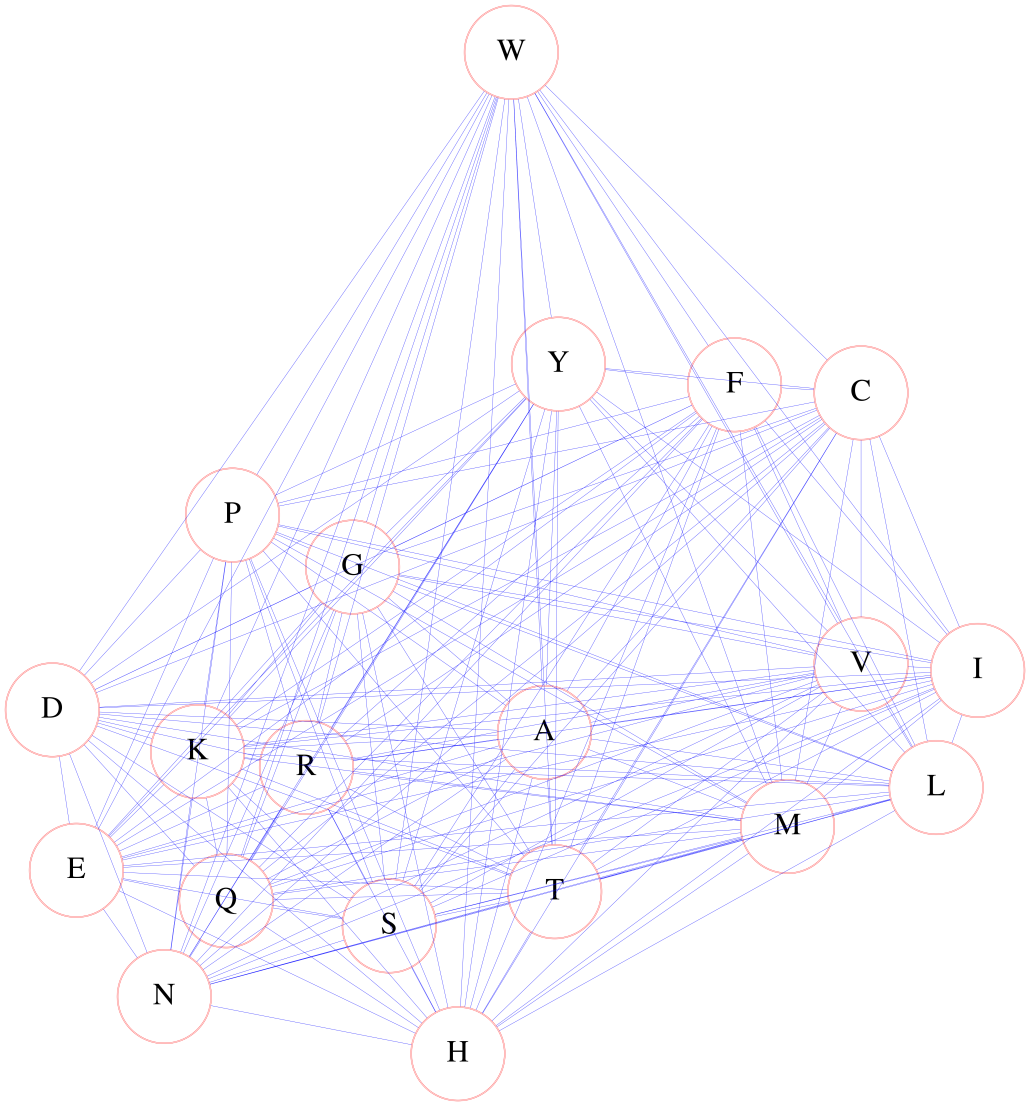

Fig. 3.4: Grafo de distancia de aminoácidos. Los ejes no representan la distancia exacta entre los aminoácidos.

Como vimos anteriormente, los compresores trabajan en la búsqueda de patrones y en particular buscan cadenas repetidas. Por lo tanto, podríamos ayudarlos en su tarea si, por ejemplo, reemplazáramos los aminoácidos parecidos por un aminoácido candidato. Continuando con el ejemplo anterior de A, S, R y T, podríamos reemplazar toda aparición de los aminoácidos S, R y T en las secuencias por el aminoácido A. De esta manera, las subsecuencias AS, AR, AT, RS, y demases, se transformarían todas en AA, lo cual haría que todas ellas luzcan exactamente igual para los compresores. Obviamente, estamos perdiendo información y quizás no queramos que los compresores las vean todas igual, por eso debemos ser cuidadosos a la hora de crear estos clusters y encontrar un aminoácido candidato para representarlos. Pero la idea a utilizar estará basada en eso: utilizando esta matriz (o grafo) de distancias entre aminoácidos, crear *clusters* que nos permitan agrupar por similitud y usarlos para luego reemplazarlos en las secuencias justo antes de comprimir.

3.0.6. Compresores para la diversidad

Si bien hemos introducido y hablado sobre los compresores, todavía no hemos aclarado qué utilidad pueden presentar ante el problema de diversidad que estamos estudiando en este trabajo. Como hemos visto a lo largo de este capítulo, los compresores tienen en común la búsqueda de subcadenas de datos similares que puedan reemplazarse o agruparse. Mediante diferentes algoritmos (como ventana deslizante o reordenamiento), son capaces de reconocer estas similitudes en las secuencias de símbolos.

Nos preguntamos entonces qué pasaría si como *input* a estos compresores le diésemos un conjunto de secuencias de aminoácidos. ¿Serían capaces estos algoritmos de encontrar las similitudes entre ellos e indicárnoslo? ¿De qué manera lo harían?

Una forma es imaginar que el *input* fuese la concatenación de todas las secuencias de aminoácidos que queremos verificar. Debido a que el tamaño de ventana de los compresores puede tener alguna influencia, así como que no haya una separación entre las secuencias utilizadas al momento de concatenarlas, lo hacemos de manera aleatoria en cada una de las experimentaciones. Por la forma en que trabajan los compresores, mayor similitud significará mayor cantidad de subsecuencias similares, y a la vez, mayor compresión. ¿Pero cómo medir la compresión? Podríamos hacerlo con el largo del *output* del compresor.

Supongamos el siguiente ejemplo, en donde comprimiremos simplemente con diccionarios (y para simplificar, ignoraremos el almacenamiento del diccionario). Nuestras secuencias de aminoácido tendrán un largo fijo de 5 para mantenerlo sencillo también.

Secuencias	Concatenadas	Diccionario	Output	Ratio
ARNDC, ARNDC	ARNDCARNDC	1 → ARNDC	11	80 %
ARNDC, ARNDH	ARNDCARNDH	1 → ARND	1C1H	60 %
ARNDC, LKMFP	ARNDCLKMFP		ARNDCLKMFP	0 %

Tab. 3.3: Ejemplo de compresor sencillo con secuencias de aminoácidos

Intituivamente, podemos ver que a mayor similitud, mayor es el ratio de compresión. Para ver si esta intuición era acertada, hicimos unas pruebas rápidas con datos reales. Para ello, usamos un conjunto de secuencias CDR3 de receptores β de células T, ya clasificadas en base a frente qué complejo pCMH reaccionan (provenientes de VDJdb[5]). Separamos a estas secuencias en base a dicha clasificación (a cada una de estas categorías le asignamos un número de 1 a 10), e hicimos el siguiente experimento:

Para $i \setminus i \in$ [1,10]:
  scores[i] ← [ ]
  Repetir 200 veces :

categories ← Tomar i categorias al azar
sequences ← Tomar 50 secuencias al azar que pertenezcan a *categories*
seqs_concat ← Concatenar todos los elementos de *sequences*
compress_value ← ComprimirZLIB(seqs_concat)
score ← Largo(compress_value) / Largo(seqs_concat)
scores[i].append(score)
scores[i] ← Promedio(scores[i])

El experimento comienza tomando una categoría al azar, y luego 50 secuencias de dicha categoría. Las concatena, las comprime, y divide el largo del resultado de comprimir por el largo de la concatenación original (esto se debe a que no todas las secuencias originales son iguales, entonces de esta manera normalizamos). Repetimos este proceso 200 veces (tomando, en cada caso, una categoría al azar distinta, y 50 secuencias a azar), y nos quedamos con el promedio. Ese será el *score* de nuestro compresor cuando todas las secuencias pertenecen a una categoría.

Luego repetimos lo mismo pero esta vez con dos categorías. En cada uno de las 200 veces elegimos dos categorías al azar, y luego 50 secuencias de estas categorías también al azar. El *score* final en este caso corresponderá a dos categorías. El proceso se repite de igual manera hasta llegar a 10 categorías. Si la intuición está en lo cierto, en cada paso el *score* debería ser mayor pues al mezclar secuencias de distintas categorías deberían reducirse la cantidad de patrones repetidos.

Este experimento confirma la intuición que teníamos, ya que al graficar los valores se puede ver un crecimiento en el *score* a medida que sumamos más categorías al conjunto de secuencias. En otras palabras - y volviendo a nuestro problema original - a medida que sumamos diversidad, el *score* crece, lo que nos da la pauta que puede existir una relación entre la diversidad y lo que nos indican los algoritmos de compresión, y más aún, podríamos llegar a utilizar dichos algoritmos como predictores de diversidad.

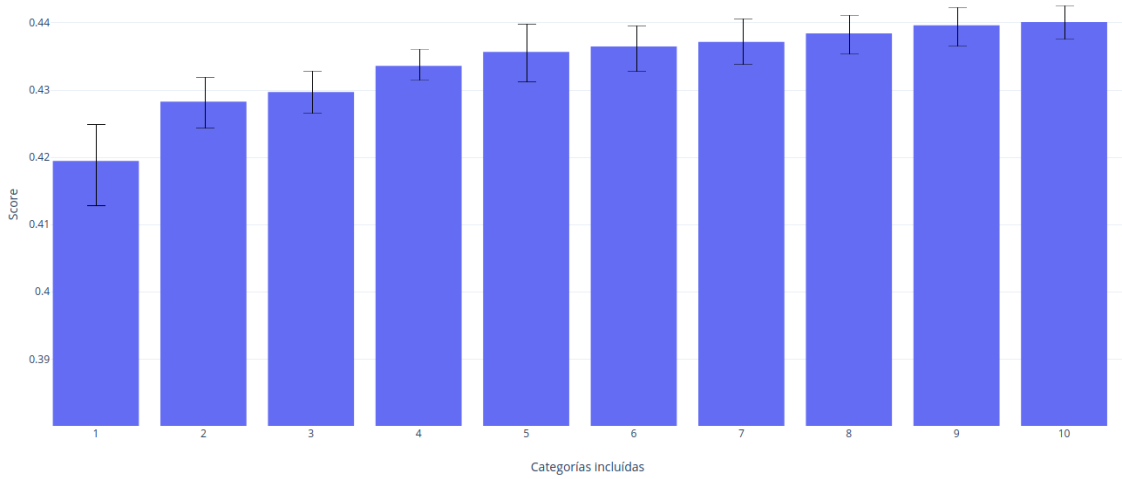

Fig. 3.5: Score de compresión de 50 secuencias acotadas por cantidad de categorías.

4. METODOLOGÍA

4.0.1. Conjunto de datos

Antes de empezar con nuestra experimentación, lo primero que debemos hacer es organizar el *dataset* con el que trabajaremos. Nuestro punto de partida será VDJdb[5], una base de datos curada, abierta y pública de secuencias de receptores de células T junto a los antígenos que reconocen. La labor de la gente detrás de VDJdb no es sólo agregar diferentes sets de datos de diferentes fuentes, si no normalizar su formato, unificar secuencias idénticas (que por diferencias de formato pueden aparecer como distintas), y compartir dichos sets con la comunidad.

En particular, nosotros trabajaremos con un subconjunto del total de los datos. Dada la naturaleza de nuestro problema (medir diversidad), lo que buscamos serán conjuntos de secuencias de TCR que reconozcan el mismo pCMH. Por lo tanto, deberemos descartar los pCMH que sean reconocidos por una cantidad baja de secuencias, pues no nos servirán para nuestra experimentación. Así decidimos ignorar aquellos pCMH que son reconocidos por menos de 70 secuencias (la elección de 70 secuencias es necesario para la creación de diferentes grupos en la experimentación, como veremos luego). También hemos filtrado por secuencias de TCR que reconocen complejos en donde el CMH pertenece a humanos o a ratones. Esto nos indica que nuestros CMH tendrán como prefijo *H-2* (ratones) o *HLA* (humanos). Si bien nos hubiese gustado trabajar solamente con humanos, no era posible por la poca data que nos quedábamos si filtrábamos por especie, por lo tanto tuvimos que utilizar tanto humanos como ratones. Teniendo en cuenta esto, nuestro conjunto de datos contendrá 6958 secuencias de TCR junto a su respectivo complejo pCMH, como indica la siguiente Tabla 4.1. Para facilitar la lectura, a todas las secuencias que estén asociados al mismo pCMH las llamaremos clase de pCMH. Así, las primeros 1931 secuencias que reconocen el complejo (HLA-B*07:02, LPRRSGAAGA) son una clase, por ejemplo. Llamaremos a este conjunto CI, por *Conjunto Inicial*.

Como vemos, el complejo pCMH dado por el CMH HLA-B*07:02 y el epítopo LPRRSGAAGA contiene 1931 secuencias, siendo el complejo reconocido por la mayor cantidad de secuencias en nuestro conjunto de datos a utilizar. En el otro extremo, tenemos al pCMH dado por H-2Kb y TVYGFCLL, que es reconocido solamente por 71 secuencias de TCR. Vale recordar que cuando nos refererimos a las secuencias de receptores de células T, estamos hablando solo del segmento CDR3 que ya mencionamos en capítulos anteriores.

Con nuestro set de datos inicial, el siguiente paso es dividir las secuencias en dos conjuntos: un conjunto de *training* y un conjunto de *testing* (aunque ambos cumplirán los dos roles, como veremos más adelante). La idea es que entrenaremos nuestro modelo con uno de estos conjuntos y validaremos el resultado con el otro. Luego repetiremos el proceso con los roles invertidos, es decir, entrenaremos con el otro y validaremos con el primero. Sin embargo, debemos ser muy cuidadodos al realizar esta división en subconjuntos. Si ambos subconjuntos son muy parecidos caeremos en el problema del *overfitting*: nuestro modelo estará entrenado para un grupo específico de secuencias, y al ser el set de *testing* muy

Group ID	Cantidad	CMH	Epítopo
1	1931	HLA-B*07:02	LPRRSGAAGA
2	895	HLA-A*02:01	GILGFVFTL
3	850	HLA-A*02:01	GLCTLVAML
4	419	HLA-A*02:01	NLVPMVATV
5	407	H-2Db	SSLENFRAYV
6	333	H-2Kb	SSYRRPVGI
7	272	HLA-A*01:01	VTEHDTLLY
8	266	HLA-A*02:01	EAAGIGILTV
9	226	HLA-A*02:01	LLWNGPMAV
10	198	H-2Db	HGIRNASFI
11	184	H-2Db	LSLRNPILV
12	150	H-2Kb	ASNENMETM
13	137	HLA-B*08:01	RAKFKQLL
14	129	H-2Db	ASNENMETM
15	123	HLA-B*57:01	KAFSPEVIPMF
16	111	HLA-B*07:02	TPRVTGGGAM
17	96	HLA-B*27:05	KRWIILGLNK
18	88	HLA-B*42:01	FPRPWLHGL
19	72	HLA-A*02:01	CINGVCWTV
20	71	H-2Kb	TVYGFCLL

Tab. 4.1: Cantidad de secuencias por pCMH

parecido, nuestra validación dará que los resultados son buenos. Pero al probarlo con un conjunto de datos nuevos, veremos que no era tan así.

Por lo tanto, como queremos que nuestro modelo sea lo más autónomo posible al conjunto de datos, nuestros conjuntos de *training* y *testing* deberán ser lo más independientes posibles. Para comenzar este proceso presentamos la definición de distancia de Levenshtein.

Introducida en 1966 por el matemático soviético Vladimir Levenshtein [6], la distancia de Levenshtein es utilizada como métrica entre cadenas de texto, e informalmente es la mínima cantidad de cambios que deben realizarse para ir de una cadena de texto a otra. Antes de definirlo, vamos con un ejemplo. La distancia Levenshtein entre la cadena *horario* y *hogares* es **3**, pues se deben hacer tres sustituciones:

- horario ⇒ ho**g**ario
- hogario ⇒ hogar**e**o
- hogareo ⇒ hogare**s**

Aparte de sustituciones, los otros cambios posibles son agregar o eliminar una letra (no es posible el cambio de posiciones). Así, pluralizar una palabra tiene una distancia de 1 (siempre y cuando sea solo agregar una s al final). La definición formal de la distancia de Levenshtein para las cadenas a y b es la siguiente:

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{si } \min(i,j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{si no.} \end{cases}$$

siendo $|a| = i$ y $|b| = j$, y siendo la función $1_{(a_i \neq b_j)} = 0$ si $a_i = b_j$ y 1 en caso contrario. La idea de la distancia $lev_{a,b}$ es recursiva, e ir *yendo para atrás*, desde el último caracter, sumando un 1 en caso de que no coincidan, y 0 en caso de que sí. La partición en casos y el tomar el mínimo permite asegurar que siempre estamos tomando el camino más corto, lo que nos da la noción de distancia. Definiremos también una distancia de Levenshtein normalizada, que será igual a la ya definida, pero donde dividiremos el resultado obtenido por el mayor largo entre las dos cadenas. Quedará definida de esta forma:

$$levNormal(a,b) = \frac{lev_{a,b}(i,j)}{\max(i,j)}$$

repitiendo otra vez que $|a| = i$ y $|b| = j$.

Lo siguiente que haremos con *levNormal* será calcular la distancia entre todo par de elementos que pertenecen a nuestro conjunto inicial de 6958 secuecias de TCR, es decir, calcularemos $levNormal(i,j)\ \forall\ i,j \in CI^2$. Una vez calculada esta distancia para todo par de secuencias, lo que haremos a continuación será crear clusters de secuencia con la siguiente condición: si $levNormal(i,j) \leq 0{,}1 \implies a$ y $b \in CI$ pertenecerán al mismo cluster . Es decir, los clusters estarán formados por todas las secuencias cuya distancia sea menor a 0.1. Por supuesto, la implicancia en la otra dirección no es cierta: que dos elementos pertenezcan al mismo cluster no quiere decir que su distancia sea menor a 0.1.

Pero bien es cierto que podemos asegurar que cualesquiera sean dos elementos, si no pertenecen a un mismo cluster, su distancia es mayor a 0.1. Y este es el primer paso hacia nuestro objetivo de generar subconjuntos independientes.

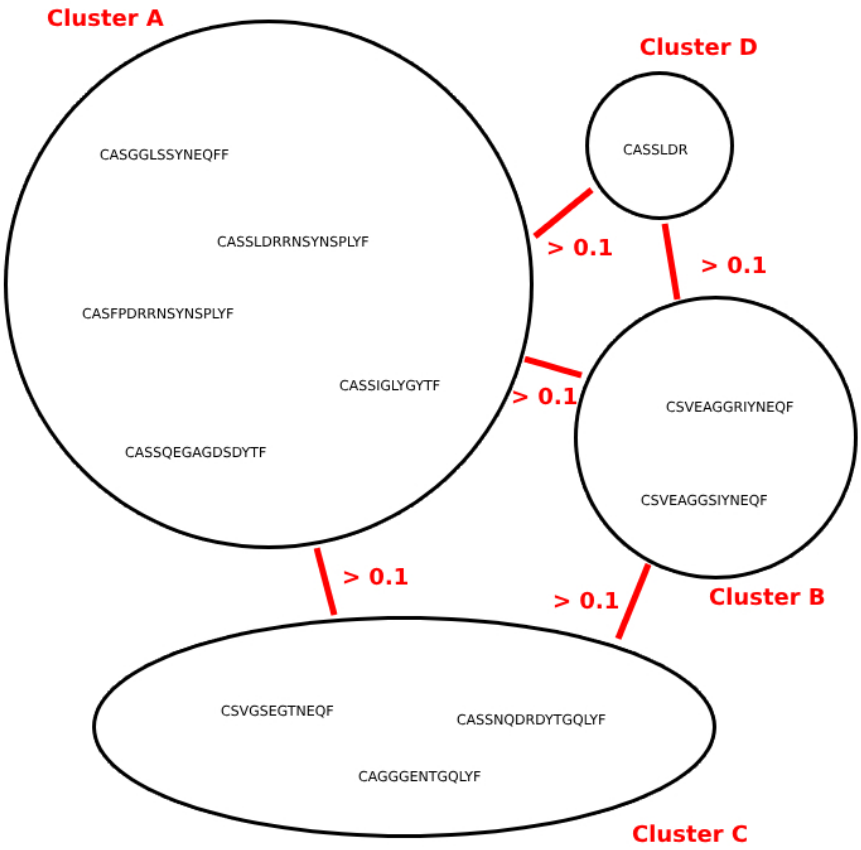

Fig. 4.1: Clusters con distancia $levNormal > 0.1$ entre ellos. Solo de manera ilustrativa, se muestran algunas secuencias y no todas ellas.

Ahora que tenemos estos clusters independientes, el siguiente paso es cómo distribuirlos entre *training* y *testing*. Si lo hiciésemos al azar, podríamos tener un problema, ya que podrían los clusters más grandes caer todos en *training* (por ejemplo); mientras que muchos clusters pequeños independientes caer en *testing*. De esta forma, habría más similitud en *training* y eso sería un problema (por la naturaleza de nuestro modelo basado en compresión).

La forma correcta de hacer esta distribución será la siguiente: ordenaremos los clusters por la cantidad de elementos que contengan, de manera descendente. Tomaremos el primero (el cluster más grande) y lo incluiremos en *training*. Luego tomaremos el segundo cluster, e irá a *testing*. Así sucesivamente con cada cluster, que será insertado en *training* o *testing*, dependiendo cuál de los dos tenga menor cantidad de secuencias. El pseudo-código es el siguiente:

```
For c \ c ∈ Clusters:
   if #Training ≤ #Testing:
      Incluir s en #Training ∀ s ∈ c
   else:
    Incluir s en #Testing ∀ s ∈ c
```

Esto nos generará los dos sets de *training* y *testing* balanceados. En efecto, al poner esto en la práctica, de nuestro set original de 6958 secuencias generamos el set de *training* y el de *testing* con 3637 y 3321 elementos respectivamente.

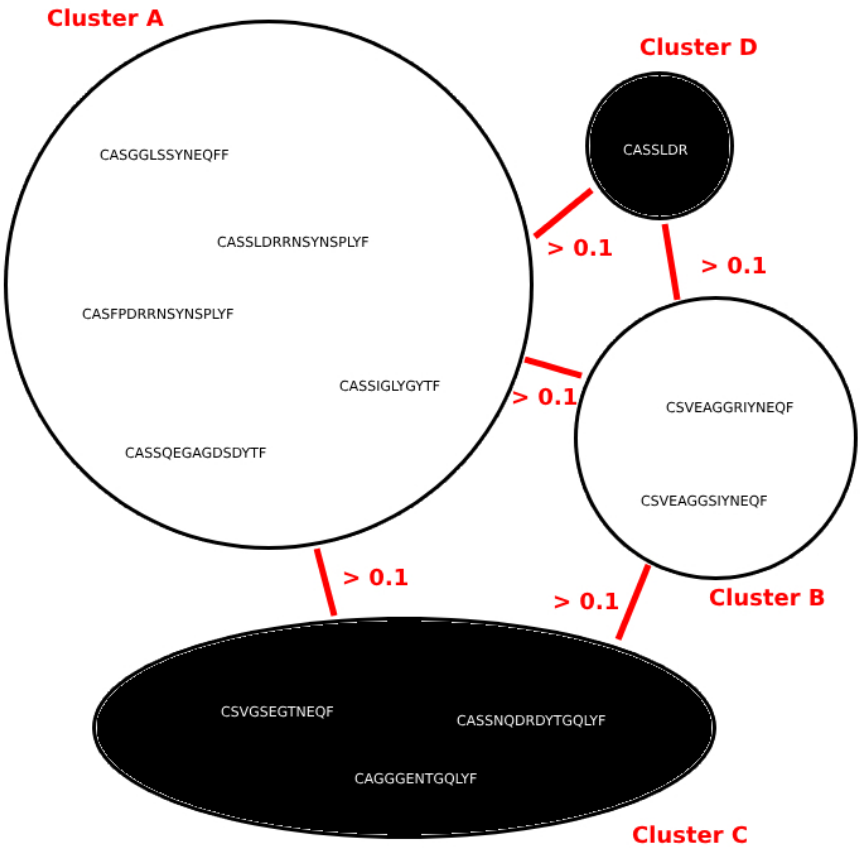

Fig. 4.2: Clusters asignados a Training o Testing. En blanco los pertenecientes a Training, en negro los de Testing. Solo de manera ilustrativa, se muestran algunas secuencias y no todas ellas.

En este punto, podemos decir que *training* y *testing* son, al menos, un 90 % independientes consdierando la distancia de Levenshtein, ya que al ser a y b secuencias tales que $a \in$ *training* y $b \in$ *testing*, $\implies levNorm(a, b) > 0{,}1$. El problema ahora que puede surgir es el opuesto, y es el de contar con secuencias muy parecidas. Si bien tener solo alguna de ellas no causaría mucho problema, sí estaría impactando en el resultado que fuera un volumen grande, porque nuestro modelo vería conjuntos con demasiada similitud y comprimiría por demás. Para evitar este problema, vamos a remover todas las secuencias redundantes, considerando redundantes a aquellas que tengan distancia $lev < 1$. Nótese que estamos tomando la función de distancia *lev* original, y no la *levNorm*, la normalizada, por lo que

estamos buscando aquellas que difieran en menos de un cambio de caracter.

La forma, otra vez, de hacer esto, es a través de clusters. Generaremos clusters dentro de *training* y *testing*, donde la distancia entre los elementos sea $lev \leq 1$. Luego, por cada cluster, nos quedaremos con un elemento (que puede ser elegido al azar), y eliminaremos todos los demás. Así, tendremos un elemento representante de cada cluster, y podemos asegurar que luego de eliminar los otros, $\forall\ a \wedge b \setminus a \in training \wedge b \in training \implies lev(a, b) > 1$. Y el razonamiento es equivalente para las secuencias de *testing*.

Después de esta reorganización y limpieza de datos, el conjunto de *training* quedó con 2586 y el de *testing* con 2570 elementos. Lo cierto es que esta limpieza no tuvo en cuenta los complejos pCMH a los que identificaba cada TCR, por lo que podría existir la posiblidad de que 1802 secuencias eliminadas reaccionaran frente al mismo pCMH. Eso sería un problema, porque podríamos estar eliminando clases enteras de secuencias y así perder pCMH (por ejemplo, si eliminárамos todas las secuencias que reaccionan frente a los pCMH menos populares, nos quedaríamos con solo 7 clases de pCMH de las 20 originales). Afortunudamente, la eliminación de secuencias se dio balanceada entre las diferentes clases - lo cual refuerza la intuición de que las secuencias parecidas pertenecen y reconocen los mismos complejos. La distribución original se puede ver en la tabla 4.1, mientras que el gráfico 4.3 muestra que todas las clases de pCMH disminuyeron en la misma proporción.

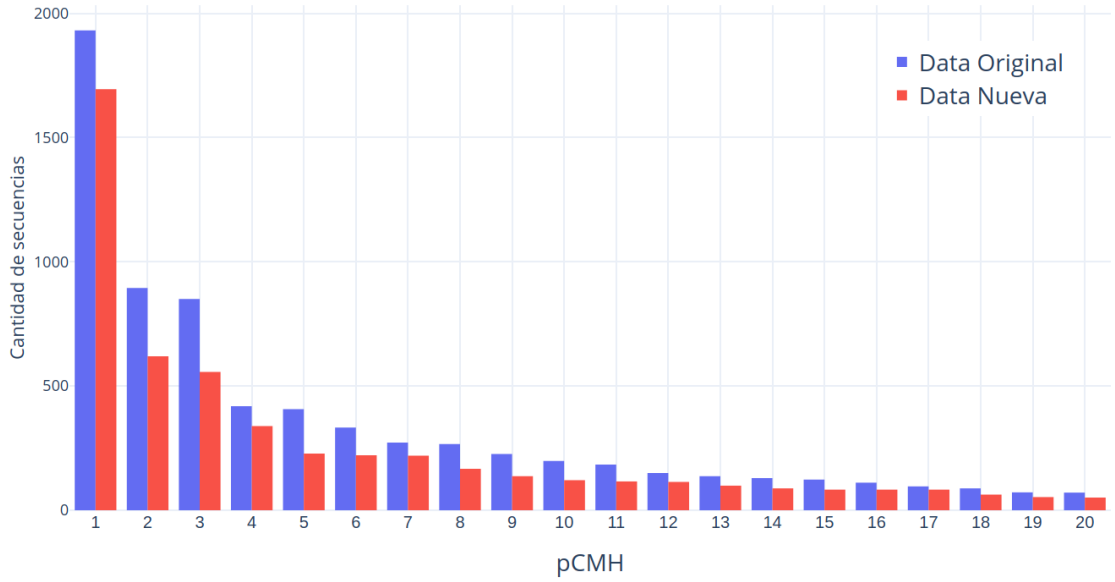

Fig. 4.3: Cantidad de secuencias por pCMH antes y después de la limpieza del dataset.

A su vez, es interesante como quedaron distribuídas las secuencias por las clases de pCMH entre el set de *training* y el de *testing*. Quisiéramos que las 20 clases de pCMH aparezcan en ambos sets. Como muestra el gráfico 4.4, las secuencias de todas las clases quedaron distribuídas equitativamente entre ambos sets, lo cual es ideal en estos casos.

Ahora que ya tenemos el set de datos listo, pasaremos a la experimentación con los modelos en sí.

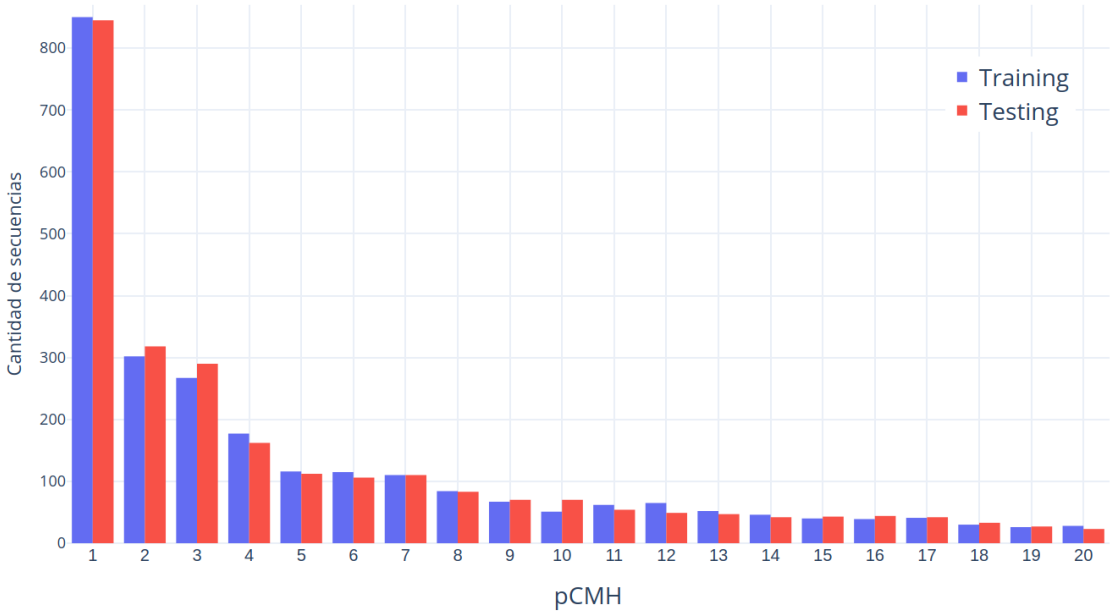

Fig. 4.4: Cantidad de secuencias por pCMH divididas entre el set de training y el de testing.

4.0.2. Método *CompreScore*

Tal como dijimos en el capítulo anterior, la idea de nuestro modelo será, dado un conjunto de TCR, devolver un *score* que servirá como estimador de diversidad de dicho conjunto. Recordemos que en este caso, para nosotros diversidad será la cantidad de clases de pCMH que las secuencias en dicho conjunto reconocerán. Si todas las secuencias reconocen el mismo pCMH, entonces la diversidad será 1, si en cambio las secuencias están divididas y reconocen cinco pCMH distintos, la diversidad será 5. En particular, podríamos definirlo de la siguiente manera:

Sea TCR el conjunto de todas los *loops* CDR3 de receptores de células T, nuestro modelo estará definido como:

$$getScore \ : \ Seqs \ \times \ Compresor \rightarrow \mathbb{R}^{+}$$

donde $Seqs \subset TCR \setminus \#Seqs = 50$ y $Compresor = \{ZLIB, GZIP, LZMA, BZIP\}$.

Es decir, nuestro modelo aceptará un conjunto de 50 secuencias de CDR3 y uno de los cuatro compresores disponibles, y devolverá un *score* que tendrá algún tipo de relación con la diversidad del conjunto que usamos como entrada.

La forma en la que haremos los experimentos tiene ciertos detalles a considerar. En primer lugar, dado que en nuestros conjuntos de *training* y *testing* ciertas clases de pCMH son reconocidas por menos de 30 secuencias de TCR distintas, consideraremos conjuntos $Seqs$ para nuestro modelo $getScore$ tal que al menos haya 5 clases de pCMHs distintas. En segundo lugar, será la forma de generar estos conjuntos $Seqs$. Lo haremos a través de una función $genSeqs$, que tendrá la siguiente signatura:

$$genSeqs \ : \ q \ \times \ Distr \rightarrow Seqs,$$

donde $Seqs \subset TCR$ y $\#Seqs = 50$, como ya vimos, y $q \in \mathbb{N}$ y $5 \leq q \leq 20$. Por último, $Distr = \{Uniforme,\ Dirichlet\}$, siendo el k de *Dirichlet* igual a nuestro valor q.

Lo que hará *genSeqs* es, en primer lugar, elegir q clases de pCMH. Luego, para cada una de ellas, elegirá una cantidad q_i, con $1 \leq i \leq q$, tal que

$$\sum_{i=1}^{q} q_i = 50$$

La forma en la que se asegurará la validez de esta sumatoria es a través del segundo parámetro de la función *genSeqs*. En el caso de que sea *Uniforme*, será sencillo: cada familia tendrá la misma cantidad (en caso de que $50 \mod q \neq 0$, se salvará haciendo que todas las clases tengan la mayor cantidad de secuencias de la forma más balanceada, asignando en principio $\lfloor\ 50\ /\ q\ \rfloor$, y luego ir asignando de manera iterativa a cada familia cada uno de los restantes). Sin embargo, en la naturaleza será muy difícil encontrar esta distribución, por lo que también consideraremos un caso menos artificial. Utilizaremos la distribución de Dirichlet, que es parte de las librerías que utilizamos en Python, y nos permitirá, dado q, obtener q números tal que su suma de 50 y dichos números sean *samples* de una distribución de Dirichlet. Así, generaremos conjuntos de secuencias en donde la cantidad de secuencias por familia no sea en todos los casos el mismo.

Teniendo ahora *genSeqs* y *getScore* podemos definir los pasos llevados a cabo en nuestro experimento de la siguiente forma:

```
Fijado Compresor ∈ {ZLIB, GZIP, LZMA, BZIP}
Fijado Distr ∈ {Uniforme, Dirichlet}
For q \ 5 ≤ q ≤ 20:
   valores[q] = []
   For i \ 1 ≤ i ≤ 200:
      seqs = genSeqs(q, Dist)
      valores[q].append( getScore(seqs, Compresor) )
   valores[q] = promedio(valores[q])
```

En pocas palabras, la idea del experimento será calcular el *score* (a través de *getScore*) unas doscientas veces para cada subconjunto de pCMH entre 5 y 20 clases, y nos quedaremos con ese promedio, creando una relación entre q (de 5 a 20, significando la cantidad de clases utilizadas para el cálculo), y *score*. Esta relación nos permite conocer *score* en función de q (o al menos un estimado). Lo que buscamos nosotros, para nuestro modelo final, será lo contrario. Por eso nuestro modelo será una inversa de esta función, que nos permitirá dado un *score*, saber la cantidad de clases de pCMH presentes en ese conjunto, que es en definitiva la diversidad que buscamos.

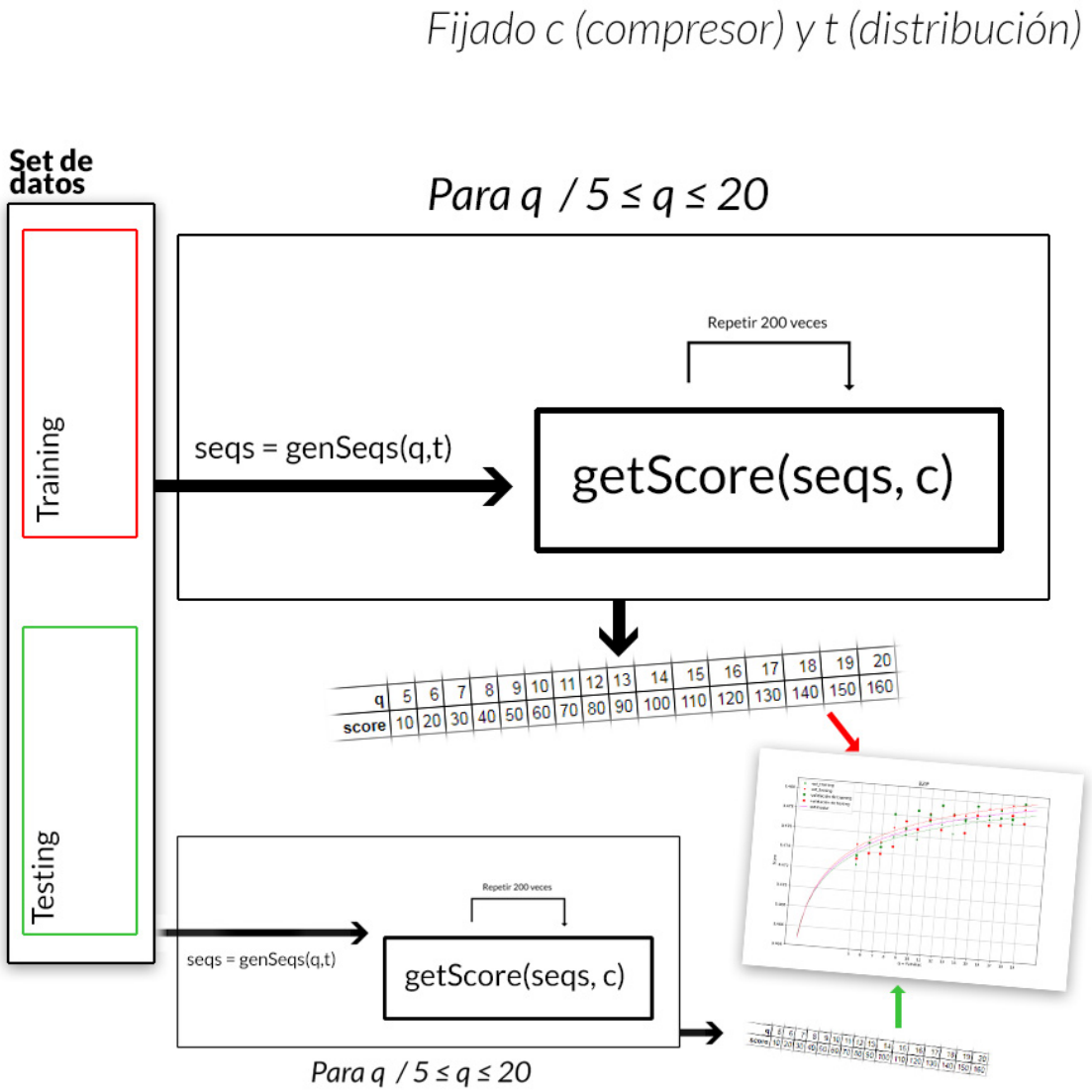

Fig. 4.5: Diagrama de experimentación. El proceso se realiza con los datos de *training*, que da una relación entre cantidad de clases de pCMH y nuestro *score*. Luego se realiza el mismo proceso con los datos de *testing*, y se comparan ambos resultados a través de los estimadores logarítmicos generados por cada uno de ellos.

Al ver los primeros resultados, elegimos interpolar nuestros datos a una función logarítmica de la forma $a * log(x) + b$. ¿Cómo es que podemos validar entonces que esta función - o su inversa, en realidad - sería una buena aproximación? Ahí es donde entran en juego los conjuntos de *training* y *testing*, y la importancia de su independencia.

Primero, los pasos del método anterior se harán utilizando el conjunto *training*. Es decir, la función *genSeqs* trabajará sobre ese conjunto. Al finalizar todas las iteraciones, obtendremos una función logarítmica (la relación entre q y *score*) que llamaremos $getScore_{training}$, y su inversa será el estimador basado en *training*, $getDiv_{training}$ (de *get diversity*, por supuesto, cumpliendo que ${getDiv_{training}}^{-1} = getScore_{training}$. Teniendo entonces $getDiv_{training}$,

podemos realizar la validación contra el conjunto de *testing*.

El proceso será similar al anterior. Para cada $q \setminus 5 \leq q \leq 20$, realizaremos 200 iteraciones en donde calcularemos el *score* y nos quedaremos con su promedio. Por supuesto, esta vez utilizaremos las secuencias provenientos de *testing*. El *score* que obtenemos luego será el *input* de nuestro $getDiv_{training}$, que nos dirá, para ese *score*, cuántas clases de pCMH debería haber. En nuestro caso ideal, $q_i = getDiv_{training}(score_i)$. Como es de esperarse, nuestro caso ideal no funcionará, así que tomaremos el error cuadrático medio de la siguiente manera:

$$ECM_{training} = \frac{\sum_{i=5}^{20}(q_i - getDiv_{training}(score_i))^2}{16}$$

Sin embargo, a la hora de haber creado *training* y *testing*, la nomenclatura usada es artificial. Solo los hemos nombrado así por el uso que le dimos, pero no por los datos que los contienen. Por lo tanto, repetiremos todo el proceso anterior (entrenamiento del modelo y posterior validación), con los conjuntos alternados. Es decir, obtendremos un $getDiv_{testing}$, generado a través de las secuencias que pertenecen a *testing*, que validaremos contra las de *training*, y esto nos dará un error $ECM_{testing}$. El error, en este caso, que consideraremos, será el promedio de los dos.

Más aún, como ahora tenemos dos estimadores, los agruparemos a ambos creando el promedio entre los dos. Si $getDiv_{training}(x) = a * log(x) + b$ y $getDiv_{testing}(x) = c * log(x) + d$, el estimador final será

$$getDiv(x) = \frac{a+c}{2} * log(x) + \frac{b+d}{2}$$

Recordemos que tendremos un estimador para cada compresor y cada distribución, por lo que luego de nuestra primera etapa de experimentación, tendremos 8 estimados distintos. Antes de pasar a los resultados, nos falta definir cómo funcionará nuestra función f, que es la que genera el *score* dado un conjunto de secuencias. Recordemos que recibe dicho conjunto (de 50 elementos, provistos por *genSeqs*), y un compresor (que podía ser *ZLIB, GZIP, LZMA* o *BZIP*). La forma de $getScore(seqs, compresor)$ será:

```
getScore(seqs, compresor):
   full_string = ¨¨¨
   For s \ s ∈ seqs:
      full_string = concat(full_string, s)
   return largo( comprimir (full_string, compresor) ) / largo
(full_string)
```

La idea es concatenar todas las secuencias que son parte del *input*, y devolver la proporción entre el largo luego de haberlas comprimido con el algoritmo *compresor*, y el largo de la concatenación (esto último es una normalización sobre el largo de la cadena). Así, volviendo a repasar la idea original, esta *getScore* nos dará un *score* más bajo cuando las secuencias sean parecidas (mayor compresión, entonces el largo será más corto), y un *score* más alto cuando las secuencias sean menos parecidas.

Ahora que hemos definido el proceso de experimentación en su totalidad, podemos ver los primeros resultados.

4.0.3. Compresión sin pérdida en conjuntos de TCRs

Como ya hemos repetido, para cada compresor hemos realizado una doble experimentación. En una donde las secuencias se han distribuído uniformemente entre las clases de pCMH, y otra donde lo han hecho en base a una distribución de Dirichlet. Los resultados para ambos son análogos, y mostramos los gráficos de la distribución uniforme.

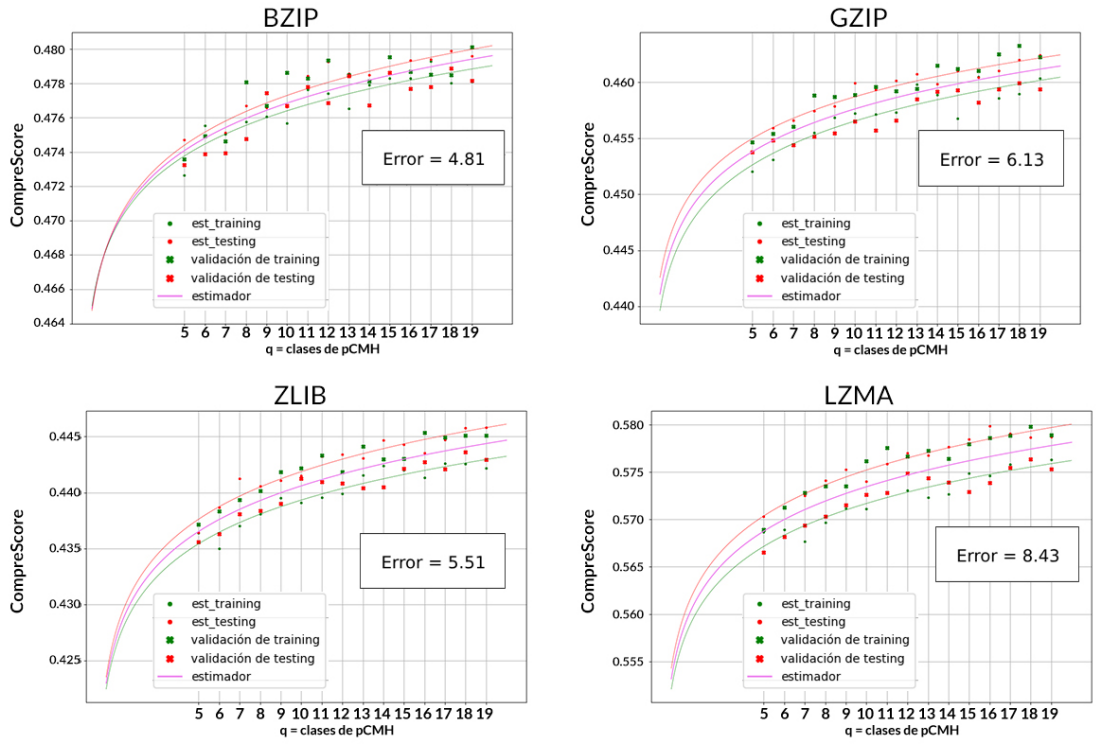

Fig. 4.6: Modelos utilizando los cuatro compresores disponibles. En verde el ajuste generado por el dataset de *training*, mientras que en rojo el ajuste generado por el dataset de *testing*. Cuanto más cerca ambas funciones logarítmicas, mejor (el promedio de ambas se ve en el estimador violeta). El error es el ERM promedio entre ambos procesos.

Mientras que con *LZMA* se ve que las interpolaciones generadas con los sets de *training* y *testing* están distanciadas, con *BZIP* se llega a notar que ambas funciones, $getDiv_{training}$ y $getDiv_{testing}$ están más cerca, lo que provoca un estimador general más exacto. Vemos también que los casos de *GZIP* y *ZLIB* son análogos al de *LZMA*.

Si bien estos resultados confirman nuestra tesis sobre la posibilidad de generar modelos que predigan la diversidad de un repetorio de células T utilizando compresores datos, nos preguntamos también si es posible mejorarlos introduciendo conceptos biológicos en ellos.

4.0.4. Compresión utilizando reducción de alfabeto basado en BLOSUM

Introdujimos en el capítulo anterior la matriz de *BLOSUM* y desarrollamos luego una matriz de distancia entre aminoácidos, la cual incluso llegamos a graficar como un grafo de distancia.

Queremos que los algoritmos de compresión ya presentados puedan sacar provecho de la información provista por BLOSUM. Nos interesaría, por ejemplo, que aminoácidos cercanos compriman mejor, ya que químicamente podrían comportarse de manera similar a diferencia de aminoácidos que se encuentran a mayor distancia. La forma de realizar eso será generando clusters de aminoácidos (basados en la distancia), y luego en las secuencias de CDR3, reemplazar todos los aminoácidos que pertenezcan al mismo cluster por el mismo símbolo. Para el compresor, así, todos los aminoácidos que estén suficientemente cerca como para pertencer al mismo cluster serán el mismo aminoácido, y podrá sacar provecho encontrando más patrones cuando haya similitud biológica. Este proceso se conoce como *reducción de alfabeto*.

La forma de generar dichos clusters será, otra vez, con el método de *cutoff*. Dada las distancias que existen entre los aminoácidos, intentamos barrer todas las posibilidades generando clusters con cutoffs enteros, entre 6 y 11 inclusive (nos movimos entre estos valores pues la máxima y mayor distancia entre aminoácidos se encuentran entre esos valores). Para comparar cuál de todos los conjuntos de clusters es más efectivo utilizamos el coeficiente de Silhouette[7], una medida para interpretar la consistencia de los clusters generados.

Silhouette asigna un valor a cada elemento de los clusters, entre -1 y 1, donde un valor alto significa que el elemento está bien ubicado en su cluster, mientras que un valor bajo significa que fue pobremente asignado. Como vemos en el gráfico 4.7, con el cutoff de valor 8 llegamos a conseguir el valor de Silhouette más alto, lo que significa un buen corte entre el conjunto de aminoácidos.

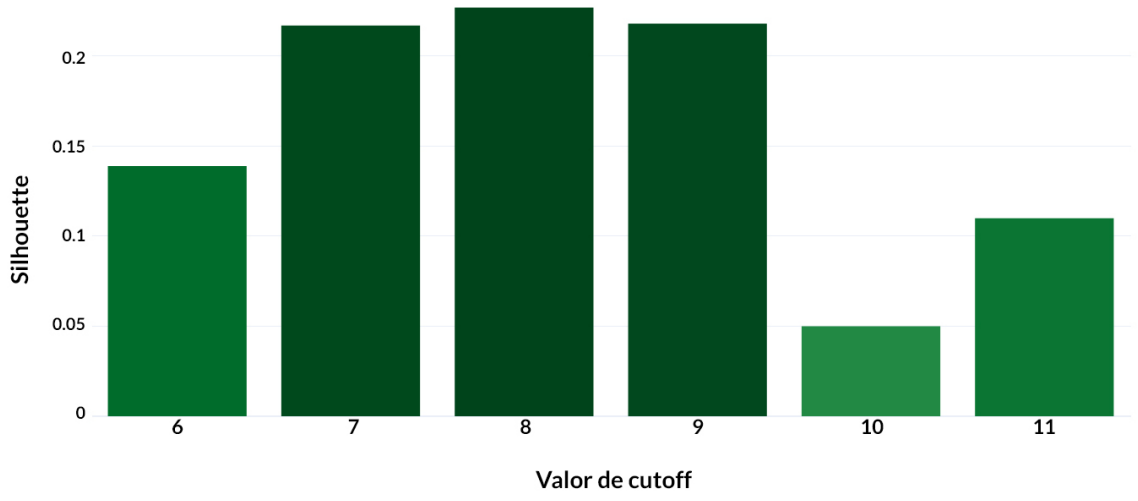

Fig. 4.7: Promedio del valor de Silhouette para cada valor de cutoff. Nótese que el valor más alto es con corte = 8.

Al generar los clusters con un cutoff de valor 6, el promedio de Silhouette de sus elementos

era de apenas 0.13. Con un cutoff de 7 subía hasta 0.217, mientras que con cutoff de 8 llegaba a su máximo, 0.227. A partir de ahí comienza a bajar, llegando a 0.1 con los clusters generados con un cutoff de corte 11. Con este el algoritmo de cutoff con corte 8, generamos 9 clusters. Los aminoácidos C, G, H, P y W quedan como clusters independientes, mientras que I, L, M y V forman un cluster; K, Q, R, D y E forman otro; N, T, S y A forman el octavo; y F e Y forman el último, como se ve en la figura 4.8.

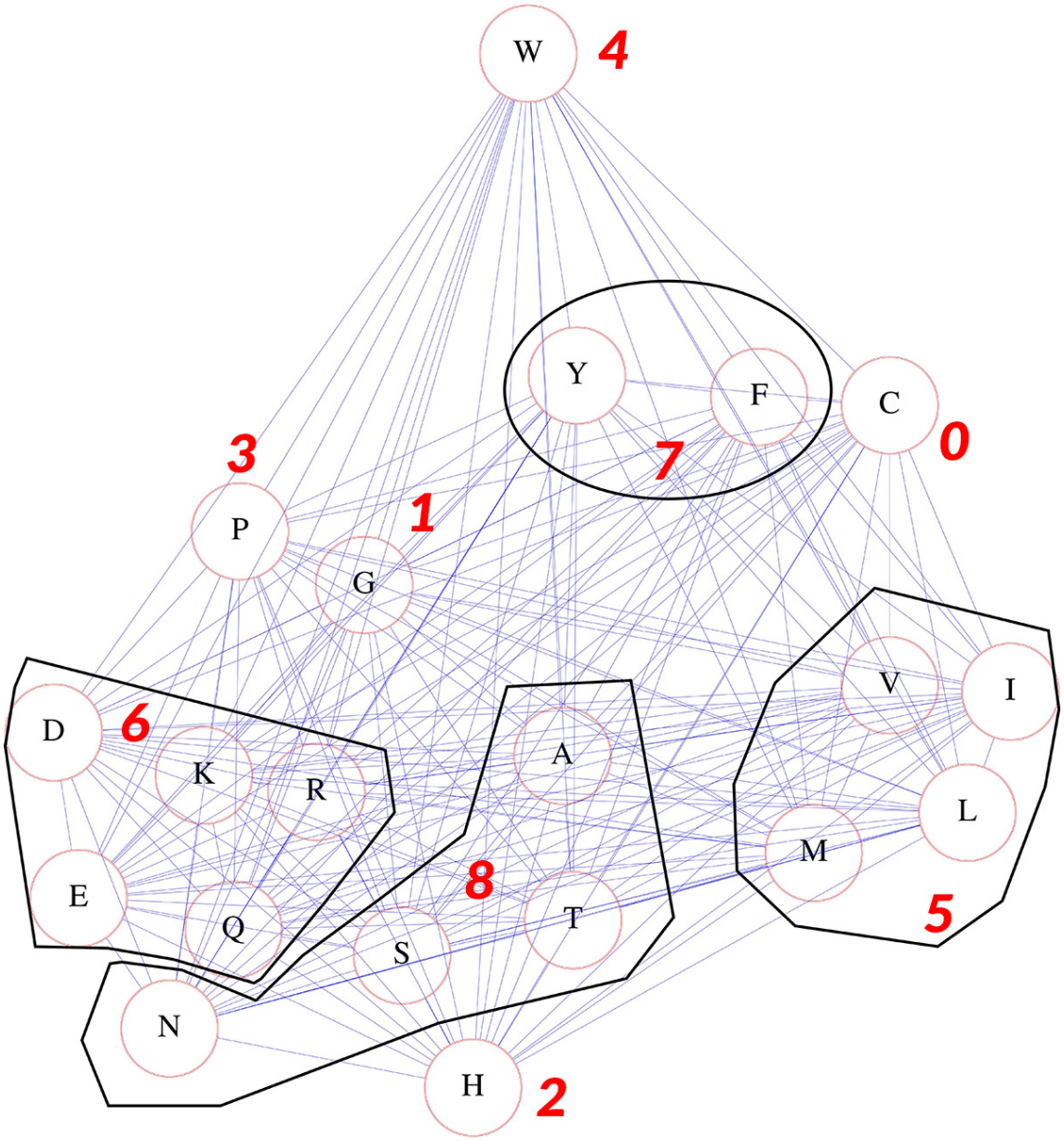

Fig. 4.8: Aminoácidos agrupados por clusters

Teniendo estos clusters, ahora nuestro experimento se desarrollará de la misma manera, con la única diferencia de nuestra función $getScore$. La reemplazaremos por $getScore'$, que estará definida como $getScore'(seqs, compresor) = getScore(blosum(seqs), c)$. Por su parte, la nueva función $blosum$ recibirá un conjunto de secuencias y devolverá otro

conjunto de secuencias, habiendo reemplazando cada caracter de cada secuencia por el ID de cluster al que corresponde. Por ejemplo, la cadena *CILKMFV* se convertirá en *0556575*, mientras que la cadena *CIMQVFV* también se convertirá en *0556575*, y aunque originalmente sean distintas, nuestro compresor las verá como la misma cadena.

Habiendo ya definido el reemplazo que haremos en base a la similitud calculada por la matriz de *BLOSUM*, volvimos a realizar nuestros experimentos de la misma manera que en el caso anterior - y los resultados se han vuelto más interesantes. El error cuadrático se ha hecho más chico en todos los casos, lo que significa que matemáticamente ha habido una mejora al introducir la similaridad entre aminoácidos.

Compresor	**Error en dist. uniforme**	**Error en dist. Dirichlet**
BZIP	4.52	3.97
LZMA	4.15	6.27
ZLIB	2.31	3.07
GZIP	2.15	2.83

Tab. 4.2: Error de cada compresor con cada distribución, utilizando reemplazo por Blosum

Las mejoras han sido varias. Por ejemplo, con *LZMA*, el error bajó en un 50 %, de 8.43 a 4.15. Dejando de lado *LZMA* con distribución de Dirichlet, que tiene un error de 6.27, ninguno de los errores supera 4.5, mientras que en el caso original sin haber usado *BLOSUM* era lo opuesto: sólo uno de los ocho errores era menor a 4.5.

No obstante, lo más sorprendente es el desempeño de GZIP, que pasó de un error de 6.13 a 2.15, disminuyendo a un 35 % de su error original. Cuando vemos el gráfico que ilustra los dos estimadores, vemos el por qué (fig. 4.9):

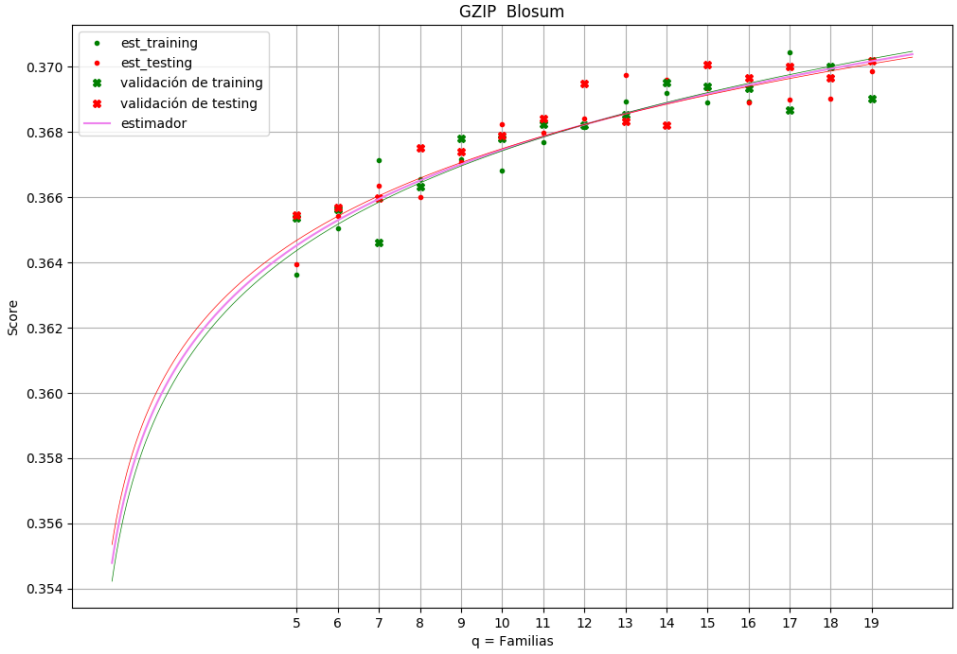

Fig. 4.9: Modelo utilizando GZIP con reemplazo por Blosum, utilizando la distribución uniforme para asignar secuencias a cada una de las clases.

Ambos estimados logarítmicos son casi idénticos. En particular, para *GZIP* con *BLOSUM*, $getDiv'_{training}(x) = 0{,}0044 * log(x) + 0{,}3572$, y $getDiv'_{testing}(x) = 0{,}0040 * log(x) + 0{,}3581$, lo que nos da finalmente

$$getDiv'(x) = 0{,}0042 * log(x) + 0{,}3577$$

con la distribución uniforme (la de la figura 4.9). Para la distribución Dirichlet, la función que ajusta es similar: $getDiv'(x) = 0{,}00318 * log(x) + 0{,}3579$.

Esto último no es sólo un indicativo de que el introducir el concepto de *BLOSUM* a nuestro estimador lo ha mejorado lo suficiente, si no que también, el hecho que haya dado (casi) el mismo modelo en ambos casos, aún cuando fue entrenado con sets independientes que tienen una distancia mínima de Levenshtein normalizada de 0.1, es también un indicio que nuestro modelo podría ser indepediente del conjunto de datos utilizado para entrenarlo (podría, entonces, funcionar con otras especies).

4.0.5. Validación del modelo frente al azar

Antes de pasar a la siguiente etapa, para comprobar y asegurarnos que el modelo no funciona por simple azar, haremos dos chequeos de control. Nos preguntamos, en primera instancia, qué sucedería si todas las secuencias de TCR fuesen exactamente iguales - es

decir, que todas tuviesen al mismo y único aminoácido. Una forma de plantear esto, siguiendo con la idea de clusters de BLOSUM, sería realizar el mismo método anterior (el utilizado para generar clusters con cutoff), pero utilizando un cutoff muy grande. Es decir, nuestro nuevo valor podría ser $cutoff = max(dist(i,j)) + 1, \forall i,j \in Aminoacidos$. De esta manera, todos los aminoácidos caerían en el mismo cluster y así serían siempre reemplazados por el mismo número. Por ejemplo, luego de pasar por esta transformación, ambas cadenas de aminoácidos *CILKMFV* y *CIMQVFV*, se convertirían ambas en 0000000, y en realidad esto sucedería para todas las cadenas. Intuitivamente, nuestro *getScore* no podría diferenciarlas y su valor debería ser casi constante.

Luego de hacer este experimento podemos confirmar que esta intuición es válida. Como vemos en el gráfico 4.10, en donde mostramos el resultado con el compresor GZIP (pero los demás son similares), vemos que no hay una clara relación entre la cantidad de clases de pCMH elegidas y el score obtenido. Nótese que en la figura 4.10 el eje Y (que muestra el *score*) varía con un orden de magnitud menos que en la figura anterior, por lo que se muestra casi constante si lo comparamos con el otro.

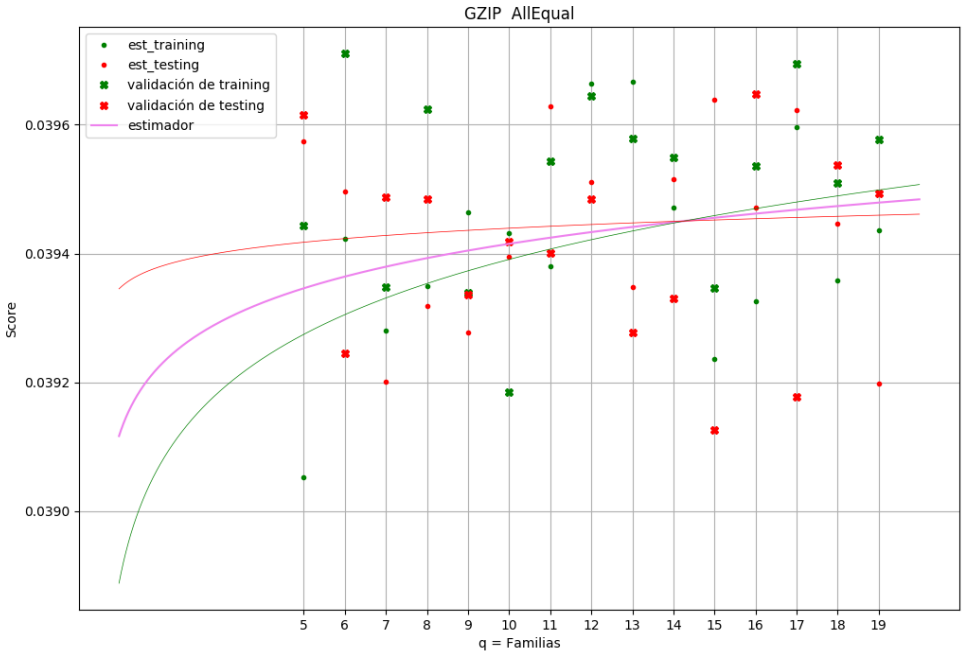

Fig. 4.10: Modelo utilizando GZIP con todas las cadenas iguales

Pasado este control, el segundo a realizar será habiendo roto la relación entre cada secuencia y la clase de pCMH que reconoce. Asignamos al azar una clase de pCMH distinta a la original. De esta manera, cuando estemos calculando el *score* para 5 familias, en realidad lo estaremos calculando para una cantidad distinta pues las secuencias obtenidas podrían pertenecer a otra cantidad en los datos originales y verdaderos. La intuición nos dice que posiblemente los resultados obtenidos sean similares a los del primer control, donde no se ve ninguna relación entre el *score* y la cantidad de familias. Si bien en el primer control

tenía sentido que el valor del *score* sea constante, acá podría ser otro tipo de relación pues en definitiva las secuencias son obtenidas al azar.

Hemos hecho este control con los cuatro compresores y naturalmente, en todos ellos el resultado fue similar. Mostramos para el caso BZIP (figura 4.11), en donde podemos ver que la distribución del *score* no sigue ningún patrón claro, y los valores suben y baja sin seguir ninguna relación clara con la cantidad de familias.

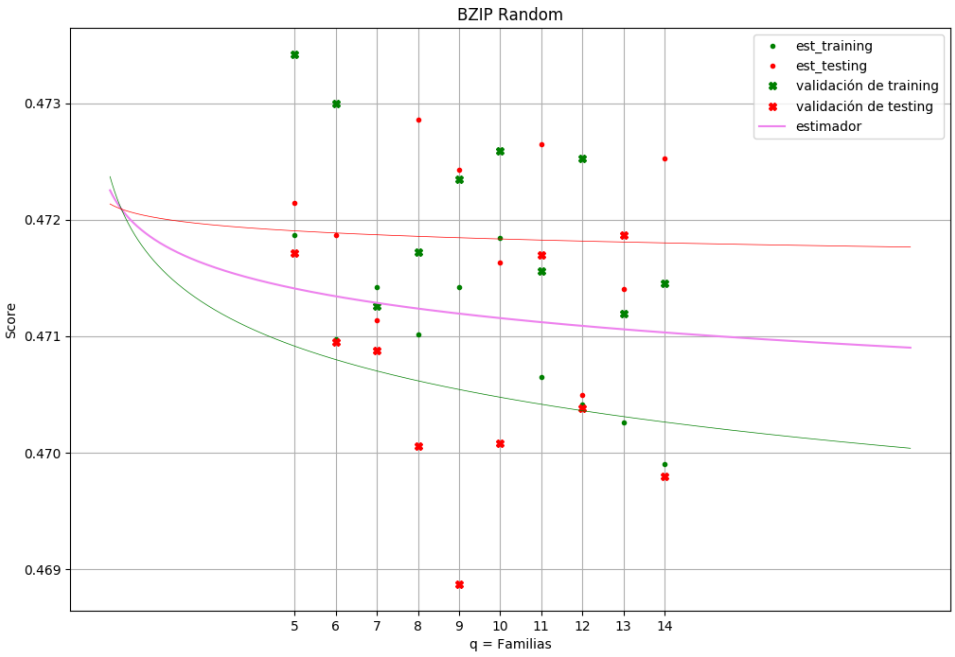

Fig. 4.11: Modelo utilizando BZIP con las relaciones entre secuencias y pCMH randomizadas.

Estos dos controles refuerzan otra vez que nuestro modelo sí parece estar tomando en cuenta *features* de las propias secuencias y su relación con los pCMH, lo que nos da mayor confianza en el mismo. Teniendo entonces nuestro modelo *getDiv* funcionando y habiendo pasado estos controles, el siguiente paso será intentar aplicarlo a datos provenientes de distintas etapas de pacientes reales, en su estadíos pre y post vacunación.

5. APLICACIÓN

Como vimos en los capítulos anteriores, los compresores de texto son capaces de dar un estimado de la diversidad de receptores de células T. Más aún, también vimos que la introducción de similitudes entre aminoácidos, como ha sido la matriz BLOSUM y su uso para el reemplazo de aminoácidos, ha mejorado esa estimación. Por eso creemos que nuestro estimador podrá ser útil a la hora de analizar cambios en el sistema inmune (mayor o menor diversidad a lo largo del tiempo). Así es como intentaremos aplicar nuestro modelo a dos casos diferentes de la misma área, donde a su vez lo compararemos con métricas existentes.

5.0.1. Compresibilidad pre y post vacunación

El primer caso en donde aplicaremos nuestro modelo es frente a un set de datos proveniente de dos pacientes, los cuales llamaremos paciente A y paciente B. Los datos a utilizar fueron extraídos de los estudios [8] y [9], que también nos permiten saber, por ejemplo, que el paciente A es un hombre caucásico de 51 años, mientras que el paciente B es una mujer caucásica de 50 años, ambos con un melanoma cutáneo. Ambos pacientes recibieron vacunas en diferentes estadíos del proceso, y pudimos acceder a una secuenciación de una muestra sanguínea en el momento pre vacunación, y dos estados posteriores a la vacunación (cabe destacar que los pacientes recibieron distintas dosis a lo largo del proceso, por lo que los dos estados post-vacunación tuvieron dosis intermedias).

Los datos de los estudios mencionados previamente - en cada una de estas etapas - son *samples* sanguíneos en donde sabemos los clonotipos que aparecieron en cada una de estas muestras junto a la cantidad de apariciones. Por ejemplo, sabemos que en el estado post-vacunación 2 del paciente B, el clonotipo *CASSTRGSWNTGELFF* aparece 3001 veces, mientras que *CASSPQTAGGLETQYF* solamente 91 veces. Debemos ser cuidadosos, no obstante, porque esto es solo una muestra sanguínea, una fotografía acotada de su sistema inmune y no debemos tomarlo como en su totalidad.

Para salvar esta particularidad de no contar con toda la información del paciente, haremos un experimento simulado. Tomaremos los datos disponibles como si fuesen en realidad la sangre total del paciente, y extraeremos de allí muestras más pequeñas, de forma azarosa, que utilizaremos para nuestros experimentos de manera repetida (todas muestras distintas, que en realidad serán *subsamples* de la sangre total del paciente). En cada uno de nuestros experimentos tomaremos 4000 clonotipos que, vale aclarar, pueden estar repetidos. Es decir, volviendo al ejemplo anterior, *CASSTRGSWNTGELFF* aparece 3001 veces, por lo que en uno de nuestros *subsamples* podría aparecer esta secuencia 3001 veces y las restantes 999 secuencias pertenecer a otra (en ese caso, nuestro *subsample* tendría solo dos clonotipos diferentes). Repetiremos, entonces, este proceso 250 veces y nos quedaremos con el valor de la media del índice de Shannon y del *score* provisto por nuestro modelo, aplicado, como dijimos, en cada una de las 250 veces a 4000 secuencias de receptores de células T.

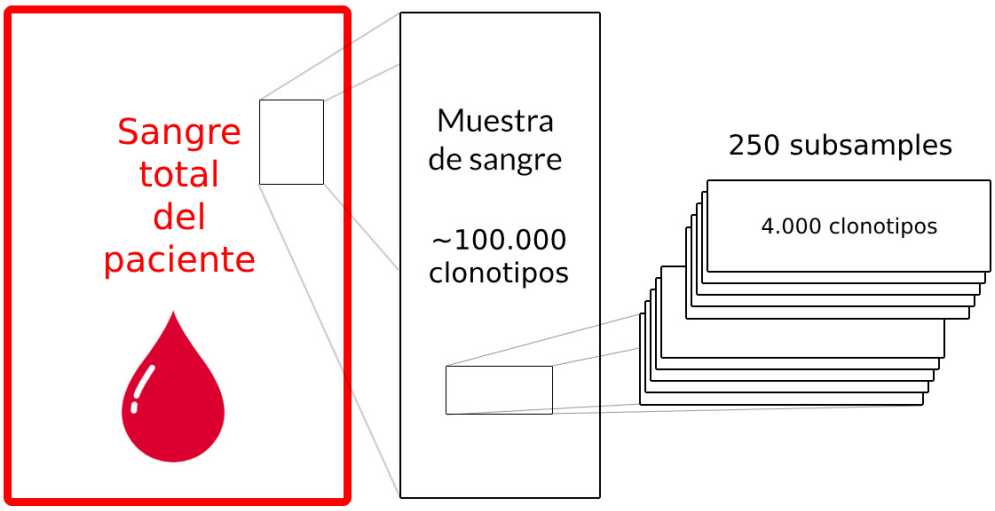

Fig. 5.1: Experimento artificial para utilizar el sample de los pacientes como si fuese la sangre total.

Al hacer este experimento y analizar los resultados vemos que el índice de Shannon muestra un cambio a través de los distintos estadíos del paciente A y del paciente B, mientras que pareciera que la media de *CompreScore* no cambia tanto. Para el paciente A, el índice de Shannon crece desde el estado PRE al POST 1 (con un p-value < 0.0001), y luego vuelve a bajar desde el estado POST 1 al POST 2, también con un p-value < 0.0001. También con un p-value similar confirmamos que el índice de Shannon baja si comparamos el estado PRE contra el estado POST 2. Lo que sucede con el paciente B es similar. El valor obtenido con Shannon baja al calcularlo desde el PRE al POST 1, con un p-value también muy bajo, aunque no es tan claro el cambio entre las etapas POST 1 y POST 2 (Figura 5.2).

Por otro lado, nuestro *score* de compresión no cambia tanto entre las distintas etapas de cada uno de los pacientes. Si bien podemos observar *outliers* que se alejan de los valores medios, y los p-values son bajos para las relaciones PRE $<$ POST 1 y POST 2 $<$ POST 1, no parece haber una clara correlación entre *CompreScore* y el índice de Shannon. Podemos tomar esto como algo positivo: nuestro modelo agrega una nueva coordenada a analizar, ya que al ser independiente de las probabilidades de aparición de los CDR3 podemos tomarlo como información nueva que provee del conjunto de datos. Sería entonces, nuestro modelo, una posible nueva medida de diversidad que provee una mirada alternativa sobre la diversidad y cómo cambia ésta en diferentes estadíos de paciente, sobretodo en pacientes que han sido sometidos a vacunas que afectan su sistema inmune. También nos da la pauta que las diferentes especies de clonotipo no cambian en las diferentes etapas de vacunación que hemos analizado.

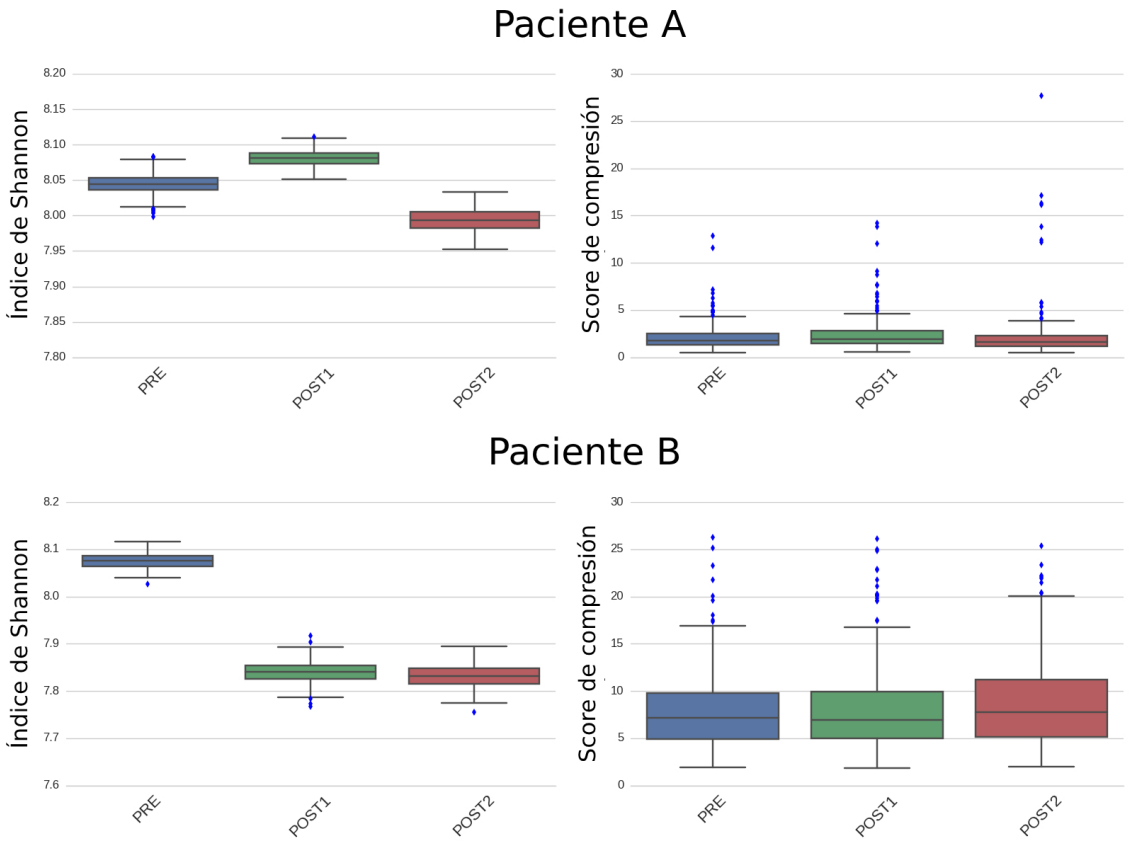

Fig. 5.2: Índice de Shannon y Score de Compresión para cada paciente y sus diferenes estados.

5.0.2. Compresibilidad en función de la edad

En nuestra segunda aplicación intentaremos ver si existe alguna relación entre el valor de *CompreScore* sobre un conjunto de receptores de células T y la edad de los pacientes. Para verificar esto utilizaremos los datasets disponibles, proveniente de [11], en el que contaremos con información de 79 pacientes distintos de entre 0 y 103 años de edad. La información de cada paciente es presentada de manera similar a la aplicación anterior: tenemos un *sample* de su sangre total, por lo que aplicaremos la misma metodología explicada en el paso previo. Tomaremos, de este *sample*, *subsamples* que utilizaremos para calcularle el valor de *CompreScore* y ver si así podemos obtener una nueva métrica sobre la diversidad de los clonotipos presentes.

Una vez calculado el *CompreScore* para cada paciente, los agruparemos en cuatro grupos etarios: los menores de 30 años (jóvenes), entre 30 y 50 años (adultos), entre 50 y 70 años (ancianos) y mayores a 70 (longevos). Esta segmentación es igual en el estudio original, por eso decidimos mantenerla. Los resultados están presentados en la figura 5.3.

El gráfico pareciera mostrar una tendencia en bajada, es decir, a medida que sube la edad de los pacientes, su *CompreScore* es cada vez más pequeño, lo que podría darnos la pauta de que su diversidad de clonotipos decae con el avance de la edad. Podemos atinar a decir que en el rango etario de jóvenes, el valor de *CompreScore* es mayor al de adultos (p-value

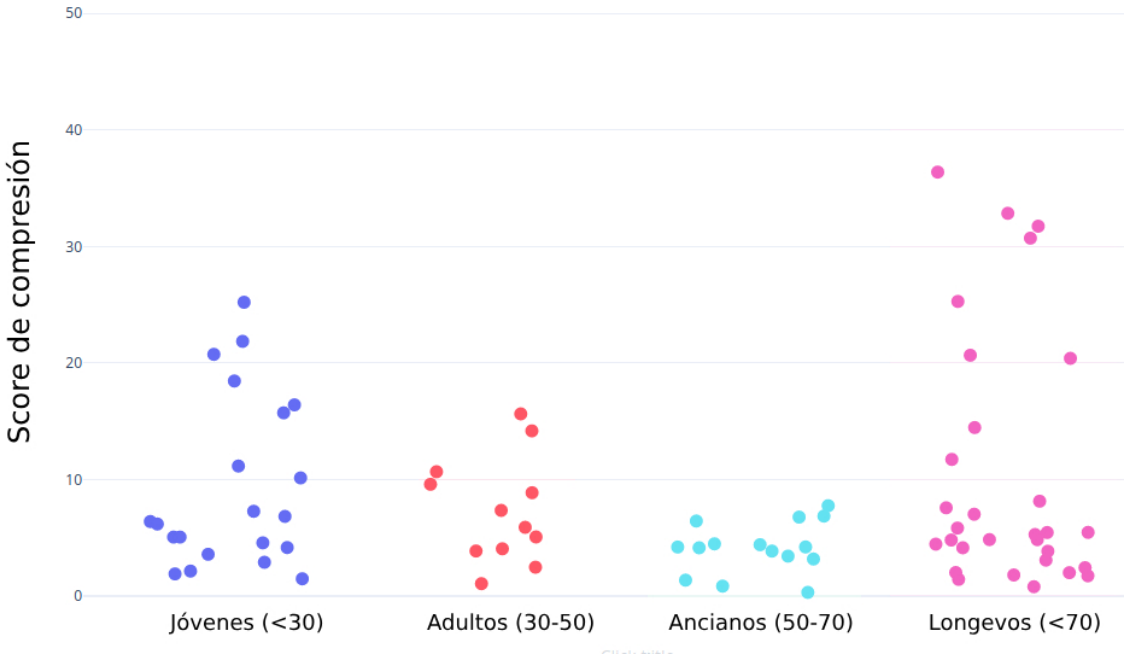

Fig. 5.3: Score de Compresión para cada paciente segmentado por rango etario

de 0.33), y a su a vez el de adultos es mayor al de ancianos (p-value de 0.07), aunque como vemos, los valores de p-value no son tan bajo como quisiéramos.

De todas formas, lo que llama la atención del gráfico son los *outliers* que se presentan en el rango de los longevos. Se ve un salto entre los *CompreScore* generados, ya que varios pacientes tienen un score por debajo de 10, sin embargo 9 pacientes tienen valores altos - con *scores* superando incluso los 30 puntos y los valores más altos del rango de los jóvenes, lo que parece extraño pues no cumple con la tendencia en baja que estábamos tratando de ver.

El *CompreScore* acompaña la noción presentada en [10], en donde se discute que a mayor edad, decrece la diversidad del repertorio de células T, y se hace una medición directa de la diversidad utilizando el método de secuenciación. En dicho estudio confirman este descenso de diversidad, pero notan algo interesante: decrece linealmente a los 40 y a los 70 años, pero alrededor de los 82 años crece linealmente si se compara contra la diversidad presente a los 62 años. Si bien no podemos afirmar que los *outliers* que aparecen estén relacionados a esa eventualidad presente en el otro estudio, sí quizás exista una relación y esa anormalidad está siendo capturada por el *CompreScore*.

Aunque no son completamente concluyentes, ambas aplicaciones muestran resultados interesantes. La primera de ellas refuerza el concepto del *CompreScore* como una nueva medida para la diversidad de repertorio de células T, introduciendo información independiente respecto a otros índices conocidos como el de Shannon o el de Simpson. La segunda experimentación con pacientes da la impresión que *CompreScore* a su vez es capaz de identificar ciertas particularidades del paciente, más allá de la diversidad de su sistema inmune, como podría ser a qué segmento etario pertenece.

6. CONCLUSIÓN

A lo largo de este trabajo planteamos una nueva forma de medir la diversidad de un conjunto de receptores de células T, en particular la diversidad presente en la región CDR3 de dichos receptores. Nuestro nuevo enfoque intenta sacar provecho de las similitudes que hay en las cadenas de aminoácidos que identifican a cada uno de estos receptores, y a través de los algoritmos de compresión de textos, que son capaces de ver patrones, intentar agrupar a estas secuencias según su capacidad de reconocer al mismo complejo mayor de histocompatibilidad, lo cual desencadena la respuesta del sistema inmune.

Nuestros primeros intentos involucraron cuatro algoritmos de compresión distintos, y a pesar de sus diferencias, todos ellos mostraron que la intuición inicial era correcta. Aunque con algo de inexactitud, mostraban reconocer la diversidad del conjunto de receptores de células T que recibían. Introducimos luego el concepto de similitud de cadenas de aminoácidos, provisto por la matriz BLOSUM, y nuestro modelo, que decidimos llamar *CompreScore*, mejoró en su estimación. Pudimos demostrar así que al utilizar información sobre la estructura química de las secuencias de los receptores, los compresores de datos pudieron sacar provecho de esto y así dar un *score* más fiable.

Respecto a los compresores y su diferencias en particular, hemos visto diferentes resultados dependiendo si utilizábamos el reemplazo por BLOSUM con pérdida. En primer lugar, cuando no lo utilizábamos y aplicábamos nuestro método a las secuencias originales, el compresor que menor error dio fue BZIP. Es extraño pues por la forma de trabajar de BZIP, que utiliza el algoritmo de Burrows-Wheeler, no tiene en cuenta la posición ni el orden de los caracteres, si no sólo la cantidad de apariciones. Esto va contra la intuición que teníamos en donde esperábamos que los compresores puedan sacar provecho encontrando patrones, no solo de repetición de los símbolos, si no también de posición u orden. De todas formas, si bien BZIP fue el que mejor ejecución tuvo, estaba seguido de cerca por ZLIB y GZIP. De hecho, estos dos últimos fueron los que mejor desempeño tuvieron cuando fueron aplicados con reemplazo por BLOSUM. Por la forma de trabajar de estos dos compresores, tiene sentido pues ZLIB (y GZIP, que es una versión optimizada de este) no solo tiene en cuenta cantidad de apariciones, si no posiciones y orden, así como también secuencias consecutivas, por lo que es capaz de descifrar patrones más interesantes de nuestras secuencias. Que hayan trabajado mejor con reemplazo BLOSUM que sin él puede deberse a que en el reemplazo se hayan generado patrones más largos dentro de la secuencia (pues ahora el alfabeto de símbolos se reduce y hay más posibilidades de que aparezcan subsecuencias en las cadenas), y antes esto no sucedía. A su vez, que GZIP haya funcionado mejor que ZLIB puede deberse a que GZIP almacena *metadata* del objeto a comprimir de manera aparte, algo que es descartado al tomar el largo de la cadena comprimida (como hacía nuestro modelo), por lo que no influía al capturar el *CompreScore* final.

Al probar nuestro modelo con datos de pacientes - y ya no con los *datasets* de *training* y *testing* - hemos visto que también ha arrojado resultados interesantes. En primer lugar, *CompreScore* es una métrica nueva e independiente de las ya conocidas, como Shannon o Simpson, para medir diversidad. Vimos que al analizar diferentes etapas de vacunación, las conclusiones que daban cada uno de los índices eran distinta, lo que proveía una nueva

coordenada de información a la que ya daba Shannon. Más aún, cuando utilizamos el modelo frente a pacientes segmentados por su rango etario, demostró haber cierta relación entre la diversidad reconocida y la edad del paciente, lo cual genera más preguntas. ¿Será posible que este modelo sea capaz, no sólo de brindar una cota inferior respecto a la diversidad del repertorio de células T, si no también información sobre el rango etario para el paciente?

Por supuesto, el modelo *CompreScore* lejos está de tener un error cuadrático medio nulo, pero aún tiene espacio para ser mejorado. Por ejemplo, así como la introducción de las similitudes provistas por BLOSUM lo han mejorado, se podría seguir por esa área y optimizar cómo es que funciona dicho reemplazo. Si bien nosotros generamos *clusters* discretos basados en la distancia entre los aminoácidos, se podría haber realizado algo *más continuo*, utilizando codificación ASCII. Por ejemplo, si tuviésemos tres aminoácidos y dos de ellos estuviesen más cerca que del otro, en vez de reemplazarlos por el mismo símbolo, podrías reemplazarlo por codificaciones del estilo 0000, 0001 y 1111, lo que daría una noción de distancia pero a su vez mantendría la unicidad de cada uno. También queremos destacar que la utilización de compresión con pérdida, por la forma en la que BLOSUM hacía los reemplazos, abre las puertas para investigar más adelante otros compresores con pérdidas que puedan realizar otros análisis funcionales de secuencias biológicas.

Esta mejora, como tantas otras, y a su vez los nuevos interrogantes planteados sobre la aplicación de compresores de texto frente a secuencias de receptores de células T, quedan abiertas para futuras investigacion y trabajos en el área.

Bibliografía

[1] Fisher, R. A., et al. *The Relation Between the Number of Species and the Number of Individuals in a Random Sample of an Animal Population*, (1943), Journal of Animal Ecology, vol. 12, no. 1, pp. 42–58.

[2] Jenkins M.K., Chu H.H., McLachlan J.B., Moon J.J. *On the composition of the preimmune repertoire of T cells specific for peptide-major histocompatibility complex ligands*, (2009), Ann. Rev. Immunol.

[3] Fainbom, L. y Geffner, J. *Introducción a la Inmunología Humana*, (2013) Médica Panamericana, Buenos Aires, 2013

[4] Miller, J.F. *Events that led to the discovery of T-cell development and function–a personal recollection*, (2004) Tissue Antigens

[5] Dmitry V Bagaev, Renske M A Vroomans, Jerome Samir, Ulrik Stervbo, Cristina Rius, Garry Dolton, Alexander Greenshields-Watson, Meriem Attaf, Evgeny S Egorov, Ivan V Zvyagin, Nina Babel, David K Cole, Andrew J Godkin, Andrew K Sewell, Can Kesmir, Dmitriy M Chudakov, Fabio Luciani, Mikhail Shugay, *VDJdb in 2019: database extension, new analysis infrastructure and a T-cell receptor motif compendium*, Nucleic Acids Research, gkz874.

[6] Levenshtein, V., *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*, (1966), Soviet Physics Doklady, Vol. 10, p.707

[7] Rousseeuw, P., *Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis*, (1987), Computational and Applied Mathematics. 20: 53–65

[8] Aris, M., Bravo, A. I., Alvarez, H. M. G., Carri, I., Podaza, E., Blanco, P. A., Mordoh, J. *Immunization With the CSF-470 Vaccine Plus BCG and rhGM-CSF Induced in a Cutaneous Melanoma Patient a TCR Repertoire Found at Vaccination Site and Tumor Infiltrating Lymphocytes That Persisted in Blood*, (2019), Frontiers in Immunology.

[9] Aris, M., Bravo, A. I., Alvarez, H. M. G., Carri, I., Podaza, E., Blanco, P. A., Mordoh, J. *Changes in the TCR Repertoire and Tumor Immune Signature From a Cutaneous Melanoma Patient Immunized With the CSF-470 Vaccine: A Case Report*, (2018), Frontiers in Immunology.

[10] Britanova, O. V., Putintseva, E. V., Shugay, M., Merzlyak, E. M., Turchaninova, M. A., Staroverov, D. B., Chudakov, D. M. *Age-Related Decrease in TCR Repertoire Diversity Measured with Deep and Normalized Sequence Profiling*, (2014), The Journal of Immunology, 192(6).

[11] Britanova, O. V., Shugay, M., Merzlyak, E. M., Staroverov, D. B., Putintseva, E. V., Turchaninova, M. A., Chudakov, D. M. *Dynamics of Individual T Cell Repertoires: From Cord Blood to Centenarians*, (2016), The Journal of Immunology, 196(12).