UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño e implementación de sistemas de anotación genómica basados en computación biomolecular y biocelular y técnicas de machine learning

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alejandro Granados Bañuls

Tutor: José María Sempere Luna

2020/2021

Resumen

El proyecto trata sobre el procesamiento de datos en formato FASTA y VCF y la transformación de estos datos al formato *pvcf*. Después estos datos serán usados para la generación de un modelo de lenguaje basado en los lenguajes k-explorables en sentido estricto. Esos modelos se utilizarán para la obtención de dos anotadores diferentes, uno basado en un autómata de Watson-Crick con una capa estocástica y otro en un autómata finito determinista con una capa estocástica. El análisis y la anotación de cadenas se realizará aplicando el algoritmo de Viterbi.

Palabras clave: Información genómica, inferencia gramatical, autómatas de Watson-Crick, anotación genómica

Abstract

The project will be based on data processing with FASTA and VCF format and the transformation of that data into a pvcf format. Then this data will be used for the generation of language model based on k-explorable languages in a strict sense. These models will be used for the obtention of two different annotators, one based on Watson-Crick automata with a stochastic layer and the other based on deterministic finite automata with a stochastic layer. The parsing and the annotation of strings will be done using the Viterbi algorithm.

Keywords : Genomic information, grammatical inference, Watson-Crick finite automata, Genomic anotation

Tabla de contenidos

Índice de figuras

1. Introducción

La Inteligencia Artificial está cada vez más presente en nuestra vida cotidiana, desde los *smartphones* hasta los diagnósticos de enfermedades. Cada vez es más usada en diferentes ámbitos y diferentes dispositivos, como coches autónomos, *chatbots* inteligentes, generación de texto, procesamiento de imágenes, detección de sentimientos en textos...

Una de las ramas de la IA en la que más se está investigando es el campo del aprendizaje automático, que está dando muy buenos resultados en ámbitos del conocimiento como el procesamiento de imágenes o del lenguaje natural.

El aprendizaje automático consiste en diseñar *programas* o *modelos* que puedan aprender a partir de una serie de datos y dar unas respuestas medianamente fiables ante otros datos. Usando esto, se puede utilizar para extraer conclusiones y conocimiento de grandes *corpus* de datos de diferentes campos, tanto académicos como empresariales.

Respecto al aprendizaje automático, hay varios ámbitos de estudio, como el *deep learning*, las redes *bayesianas* o los árboles de decisión. El apartado en el que se basa este proyecto es el conocido cómo *Inferencia Gramatical*, que se basa en el aprendizaje de una *gramática* o *lenguaje formal* a partir de una serie de cadenas de símbolos, llamadas *muestras*. Esta es una aproximación al aprendizaje automático desde una perspectiva inductiva.

La *Inferencia Gramatical* tiene varios usos, uno de ellos es la investigación de estructuras gramaticales en genómica (Sakakibara, 2005), pues se puede ver el código genético, que en el caso humano es la sucesión de los nucleótidos llamados A, C, G y T, como una gran cadena que puede ser generada por una gramática. Por lo que uno de sus usos podría dirigirse a la predicción de mutaciones genómicas en base a esas premisas.

Usando esta última rama del aprendizaje automático, se propondrá un modelo para predecir mutaciones genéticas en secuencias de ADN diseñando una librería para el lenguaje de programación *python*, para que pueda ser usado por cualquier persona que lo necesite.

1.1. Motivaciones

Desde que vi en el segundo curso del grado de ingeniería informática la asignatura de TAL (Teoría de autómatas y lenguajes formales) quedé maravillado con los conceptos vistos. Una de las cosas que más me llamó la atención es la forma en la que se podía definir matemáticamente las simbologías y estructuras gramaticales de las formas de comunicación o transmisión de conocimiento, como por ejemplo los lenguajes de programación (Vaino Aho & David Ullman, 2007, 191).

Más adelante, en la especialización de computación tuve dos asignaturas que me apasionaron de la misma manera, Percepción y Aprendizaje automático.

Estas asignaturas trataban sobre el ámbito del aprendizaje automático y el tratamiento de los datos usados por ese campo.

Una vez cursadas estas asignaturas me pregunté si no habría una relación entre los conceptos de gramáticas, autómatas y aprendizaje automático. Al final di con una rama del aprendizaje automático que utilizaba esos conceptos, la inferencia gramatical.

Con esta rama de la IA se podría descubrir las estructuras gramaticales de los datos que se pudieran codificar cómo cadenas de símbolos, cosa que podría ser útil de cara a estudiarlos y poder sacar conclusiones así como un beneficio que pudiera ser usado por todos.

Uno de los posibles beneficios podría consistir en descubrir la estructura subyacente en los genomas (Sakakibara, 2005), así se podría dar una ayuda para el diagnóstico de enfermedades del campo genético a las personas dedicadas a ello.

Así que se podría plantear un modelo de inferencia gramatical que, a partir de una serie de secuencias de nucleótidos que contienen mutaciones genómicas, pueda generar una estructura capaz de predecir si se va a producir una mutación y dónde se va a producir en una secuencia cualquiera.

1.2. Objetivos

- Implementar un modelo de Inferencia Gramatical basado en lenguajes k-testables que sea capaz de predecir las mutaciones en una secuencia arbitraria .
- Anotar una secuencia genómica arbitraria a partir del modelo generado por el modelo de Inferencia Gramatical usando un autómata de Watson-Crick.
- Implementar un método que a partir de las secuencias mutadas en formato VCF y en formato FASTA permita transformar esas secuencias a un formato *pvcf*.
- Medir la precisión del modelo a partir de unos datos de pruebas.
- Implementar los métodos informáticos necesarios que permitan realizar a cualquier usuario las actividades descritas anteriormente.

1.3. Impacto esperado

Se espera que la implementación de todas las funcionalidades genere una librería que pueda ser utilizada por diferentes tipos de usuarios:

- Profesional médico: Anotar una secuencia de ADN para comprobar si hay una mutación en la secuencia.

- Usuario relacionado con la Inferencia Gramatical: Usar las clases implementadas para proyectos propios, ya sea extraer datos de un fichero FASTA, transformar las secuencias extraídas de un VCF y un FASTA o entrenar un modelo de lenguaje basado en los lenguajes k-testables.
- Usuario interesado en el autómata generado por el modelo: Consultar el resultado generado por el modelo de inferencia gramatical, que estará en un archivo en formato JSON después de ejecutarse el entrenamiento.

1.4. Metodología

El proyecto se ha dividido en varias fases para el desarrollo de las librerías correspondientes a las que se ha hecho referencia. Para la gestión de los evolutivos se ha usado la metodología SCRUM y se ha usado TDD y *python best practices* para el desarrollo. A continuación detallamos las metodologías usadas:

- SCRUM (Schwaber & Sutherland, 2016): es una metodología ágil que realiza el desarrollo de una serie de funcionalidades en fases o etapas llamados evolutivos.
- TDD (Ignacio Herranz, 2011): Es una metodología de desarrollo llamada Test Driven Development que se basa en desarrollar las pruebas (normalmente unitarias) antes que los métodos o funciones de las clases.
- Python best practices (Loria, n.d.): Es una guía para concretar la forma en la que se desarrolla un programa en python y establecer un estándar para que el programa sea eficiente y legible para otros usuarios.

1.5. Estructura

- Capítulo 2 - Mutaciones genómicas: Este capítulo habla sobre lo que son los genomas, su estructura y sus mutaciones.
- Capítulo 3 - Aprendizaje automático: En este apartado se define lo que es el aprendizaje automático de forma detallada y se mencionan varios de los tipos que existen actualmente.
- Capítulo 4 - Inferencia Gramatical: Este capítulo trata sobre los lenguajes formales y su definición, la relación con la inferencia gramatical y los usos que se le puede dar.
- Capítulo 5 - Lenguajes k-explorables y su aprendizaje: Este capítulo define los lenguajes k-explorables y el tipo de lenguaje que se usará en el proyecto.

- Capítulo 6 - Autómatas de Watson-Crick: En este apartado se define lo que es un autómata finito de WK, qué relación tiene con los lenguajes formales y el uso que se le va a dar en el proyecto.
- Capítulo 7: Propuesta de una solución para el anotador genómico: Aquí se detalla nuestra propuesta para dar solución eficiente a la anotación de información genómica relativa a las mutaciones.
- Capítulo 8 - Herramientas utilizadas: En este apartado se mencionan las herramientas usadas durante el desarrollo del proyecto.
- Capítulo 9 - Información genómica: Aquí se definen los formatos utilizados para la definición de mutaciones y secuencias así cómo los tipos de archivos usados en bioinformática.
- Capítulo 10 - Implementación: Aquí se define qué solución específica se ha realizado y cómo se ha desarrollado.
- Capítulo 11 - Experimentación: En este capítulo se expone la experimentación realizada con la solución aportada.
- Capítulo 12 - Conclusión: Este apartado comenta las conclusiones obtenidas a partir de las especificaciones.
- Capítulo 13 - Trabajo futuro: Aquí se establecen posibles pasos futuros en base a las conclusiones obtenidas.
- Capítulo 14 - Referencias

2. Mutaciones genómicas

Las mutaciones genómicas son una fuente de enfermedades, siendo muchas de ellas graves para el ser humano. Algunas de las enfermedades genéticas más comunes en España y sus síntomas son (topdoctors, n.d.):

- Fibrosis Quística: Sus síntomas pueden ser dolor causado por el estreñimiento, zona abdominal hinchada por gases, náuseas, tos, aumento de moco, fatiga...
- Enfermedad de Huntington: Sus síntomas pueden ser rigidez muscular, lentitud en los movimientos, alteraciones de la postura y del crecimiento...
- Distrofia Muscular de Duchenne: Sus síntomas pueden ser dificultad para realizar movimientos diarios simples, fatiga después de movimientos simples, andar anormal, escoliosis y lordosis...
- Academia de Células Falciformes: Sus síntomas pueden ser fatiga, falta de energía, debilidad, dificultad para respirar, palpitaciones…
- etc., etc.

Como se puede ver, las enfermedades genéticas más comunes pueden afectar de manera severa a las personas que las padecen. Por eso hay muchos ámbitos de la ciencia que se están centrando en desarrollar métodos para prevenir o tratar estas enfermedades y minimizar su impacto en la población.

Este proyecto versa sobre intentar predecir posibles mutaciones en secuencias de ADN para poder ayudar a expertos del ámbito sanitario a detectar una posible enfermedad genética, y minimizar el impacto en la persona que la pueda padecer.

Para esto, es necesario dar una definición de lo que es un genoma, sus componentes y que significa que una mutación esté presente en ese genoma.

2.1. ADN

El Ácido Desoxirribonucleico (**ADN**) es una pieza fundamental en la vida de los seres vivos. Este forma parte de un conjunto de **cromosomas** y está formado por una secuencia de de pares de **nucleótidos** en forma de doble hélice.

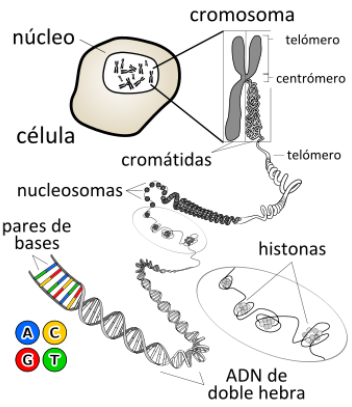

Imagen 1: Situación del ADN dentro de una célula

El ADN es el *"código fuente"* de cualquier ser vivo, pues es la estructura que contiene la información para construir los componentes de las células (las proteínas por dar un ejemplo) y las instrucciones usadas en el desarrollo y funcionamiento de todos los seres vivos (Wikipedia, n.d.). Esta información es la que permite el buen funcionamiento de la *"maquinaria"* presente en los seres.

2.2. Cromosomas

Los cromosomas son agrupaciones de ADN que están altamente organizados, contienen la mayor parte de la información genética y se encuentran en el núcleo de la célula.

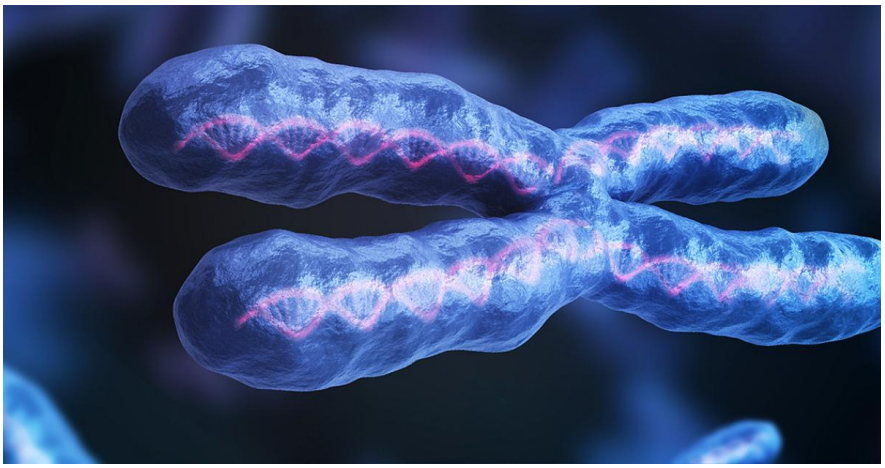

Imagen 2: Cromosoma

Cada cromosoma contiene ADN que a su vez contiene una serie de **nucleótidos**, que son la base del ADN. Un cromosoma puede contener millones de nucleótidos en su interior, por ejemplo, el cromosoma uno del genoma humano tiene aproximadamente 220 millones de pares de nucleótidos y un ser vivo puede tener varios cromosomas (el ser humano tiene 23 pares de cromosomas).

Las secuencias de ADN que se deban utilizar para realizar el proceso de predicción, que en el presente proyecto será un proceso de anotación, deberán estar acompañadas del cromosoma al que pertenecen dado que esta información es relevante.

2.3. Nucleótidos

Los nucleótidos son la pieza fundamental del ADN, pues son las bases que lo componen. Cuando ocurre una mutación, suele estar relacionada con estos últimos, pues puede pasar que uno de estos nucleótidos no sea el que toque o esté en un lugar que no le corresponde.

Las secuencias de ADN estarán formadas de estos nucleótidos, como se ha dicho en el párrafo anterior, que podrán ser representados cómo símbolos. En el caso del ser humano, el ADN está formado por 4 posibles nucleótidos llamados Adenina, Thimina, Citosina y Guanina, que serán representados como los símbolos A, T, C y G respectivamente.

Es menester mencionar que hay casos en los que los nucleótidos vienen por pares complementarios. Por ejemplo, en el caso humano hay dos pares de nucleótidos complementarios, que son la adenina y timina así como la guanina y citosina.

2.4. Mutaciones

Cuando en una cadena de ADN se produce un cambio de nucleótido se dice que se produce una mutación. Estas mutaciones pueden producir severas enfermedades en los seres vivos, pues podrían afectar a la maquinaria interna de estos. Poniendo un símil, es como si a un programa informático se le modifica una línea de código, puede que no afecte, o puede que cause severos problemas.

Las mutaciones pueden ser de varios tipos. En este caso vamos a mencionar los tipos de mutaciones en una cadena desde un aspecto de lenguajes formales más que desde el ámbito genético, pues son las bases tomadas en la solución planteada.

Supongamos que disponemos de una cadena de ADN humana:

$$S = S_0 S_1 \dots S_i \dots S_N$$

$$\forall S_j \in S,\ S_j \in \{A, C, G, T\} : 0 \leq j \leq N$$

En la que se produce una mutación. Esta mutación puede ser de tres tipos:

- La mutación de **inserción** se presenta cuando un nucleótido es añadido a la secuencia.

$$S' = S_0 S_1 \dots S_{i-1} S_i M S_{i+1} \dots S_N$$

$$M \in \{A, C, G, T\}$$

- La mutación de **sustitución** es aquella en la que un nucleótido se cambia por otro diferente.

$$S' = S_0 S_1 \dots S_{i-1} M S_{i+1} \dots S_N : M \neq S_i$$

$$M \in \{A, C, G, T\}$$

- Finalmente la mutación de **borrado** es aquella en la que un nucleótido desaparece de su lugar.

$$S' = S_0 S_1 \dots S_{i-1} S_{i+1} \dots S_N$$

Los tres tipos de mutaciones en el caso de una enfermedad específica son los que se van a intentar predecir con la realización de este proyecto.

3. Aprendizaje automático

El aprendizaje automático es una rama de la inteligencia artificial en la que cada vez se está invirtiendo más recursos por sus resultados, pues "*los ingresos del mercado global para ML como servicio (MLaaS) o plataformas de proveedores para ML totalizó 1.070 millones de dólares en 2016 y se espera que crezca a 20 mil millones de dólares para 2025*" (Deloitte, 2017).

Este aprendizaje automático consiste en diseñar un **modelo** que, a partir de una serie de **datos o muestras**, pueda predecir un valor relacionado con las **muestras** o una pertenencia a un grupo de un dato nuevo. En última instancia, se puede ver el aprendizaje automático cómo una función *Y(x)* a la que se le proporciona un dato *x* y devuelve un resultado *y* (M. Bishop, 2006, 2).

El reto consiste en hacer que el resultado *y* tenga sentido en relación con el dato *x*.

3.1. Una taxonomía

Para poder generar **modelos** que den unos resultados fiables en diferentes campos del conocimiento, el aprendizaje tiene diferentes aproximaciones que pueden ser usadas en estos campos.

En un nivel alto se pueden distinguir tres tipos de aprendizaje automático:

- **Aprendizaje supervisado**: Se basa en que los datos vienen **etiquetados** con el resultado correcto que el modelo debe dar, así que el modelo aprende a partir de esos resultados para que, a partir de un dato sin etiquetar, pueda dar un resultado en base a los vistos anteriormente.

  Esto se puede ver cómo un niño al que se le enseña una serie de imágenes de un árbol especificando que lo que se ve en la imagen es un árbol, para que cuando se le enseñe una nueva imagen diferente de las anteriores, este pueda decir si se trata de un árbol o no.

- **Aprendizaje no supervisado**: En este caso los datos van sin etiquetar, así que el modelo debe detectar patrones en los datos proporcionados para poder relacionar estos patrones con nuevos datos.

  Aquí, retomando la metáfora del niño del punto de aprendizaje supervisado, el problema podría consistir en proporcionar una serie de cubos de tres colores diferentes y que el niño los agrupe por esos colores, para que cuando se le proporcione un nuevo cubo, el niño pueda clasificarlo según el color.

- **Aprendizaje por refuerzo**: es parecido al aprendizaje no supervisado, con la diferencia que reciben un refuerzo con cada resultado para ir auto ajustándose.

Un ejemplo en relación con un niño, podría ser aquel en el que el niño dispone de una serie de fotos, y nuestro objetivo es que se distinga cual es un coche y cual no. El proceso podría radicar en que el niño vaya enseñando fotos de las que dispone y en dar una respuesta afirmativa si es coche y negativa si no lo es, para que a partir de este **refuerzo** el niño pueda aprender qué es un coche y qué no lo es.

3.2. Algoritmos

En cada tipo de aprendizaje automático se pueden clasificar diferentes **algoritmos** para generar los modelos. Los algoritmos son la maquinaria que permite aprender a partir de datos de ejemplo.

Cada algoritmo puede tener una serie de **parámetros**, que son una forma de configurar el algoritmo para generar los resultados. En el punto de lenguajes k-explorables se profundizará más en el concepto de parámetro.

Actualmente hay muchos tipos de algoritmos de aprendizaje que se basan en diferentes paradigmas. Cada paradigma tiene unas bases para diseñar sus algoritmos que normalmente se sustentan en teorías matemáticas o de las ciencias de la computación.

Por ejemplo, hay algoritmos que se sustentan en los conceptos del análisis matemático (regresión lineal, regresión logística, aprendizaje profundo), otros que se basan en las probabilidades (redes bayesianas, *naive bayes*) y otros que se basan en las teorías de gramáticas y lenguajes formales (inferencia gramatical), que es el paradigma utilizado por este proyecto.

Antes de continuar, es necesario mencionar un gran problema en relación con el aprendizaje automático, y es el problema de la **sobreestimación**, que en el capítulo de la solución adoptada saldrá a mención.

3.3. Sobreestimación

Cuando se ejecuta un algoritmo de aprendizaje automático para generar un modelo, se dice que se está **entrenando** el modelo usando los datos. Pero en este proceso puede surgir un problema que se manifiesta en la siguiente fase, que es la fase de **prueba** del modelo, pues cuando se ha generado el modelo, es necesario probarlo y sacar una métrica de su efectividad en la tarea a la que se va a enfrentar.

El problema radica en que muchas veces, después de poner a prueba el modelo, el porcentaje de acierto es muy alto, supongamos un 98%, pero cuando se saca a problemas fuera de la fase de entrenamiento y prueba, a la llamada la fase de **producción**, el acierto del modelo baja de una manera considerable, supongamos a un 30%.

Esto se puede deber a que se ha sobreestimado el modelo, es decir, se ha hecho que el modelo con los parámetros dados proporcione valores muy correctos

para los datos usados en las fases previas a producción, pero con datos reales el acierto se desplome. Esto se puede deber a que el modelo no se ha ajustado para dar buenos resultados para una gama amplia de datos, sino que se ha ajustado sólo para los datos con los que ha sido entrenado.

Recuperando el ejemplo del niño que quiere aprender a distinguir árboles del punto de aprendizaje supervisado, si sólo hemos enseñado al niño imágenes de árboles de un tipo, en una sóla estación del año y de un sólo lugar, seguramente si le enseñamos otro distinto no pueda predecir con claridad si la imágen enseñada es un árbol o no.

Por esto es necesario aplicar diferentes técnicas en la fase de **pruebas** para evitar este comportamiento.

4. Inferencia gramatical

La inferencia gramatical es un tipo de aprendizaje automático que se basa en la teoría de autómatas, lenguajes formales y gramáticas. Este tipo de aprendizaje automático aprende la estructura de una gramática a partir de una serie de cadenas de símbolos.

Antes de profundizar en los conceptos de la inferencia gramatical es necesario explicar cuáles son sus bases, es decir, los lenguajes formales, los autómatas y las gramáticas. Todas las definiciones dadas en este capítulo en relación con la teoría de autómatas, lenguajes formales y gramáticas han sido obtenidas de (J. Hopcroft, 1979).

4.1. Lenguajes formales

Un **lenguaje formal** es un conjunto de **cadenas** de **símbolos** los cuales pertenecen a un **alfabeto**.

Antes de continuar es necesario mencionar que hay un tipo de palabra especial que es la cadena vacía, una cadena que no contiene ningún símbolo. Esta cadena es denotada como λ.

Poniendo un ejemplo para explicar los términos, supongamos que tenemos un alfabeto que está formado por los símbolos a y b:

$$\Sigma = \{a, b\}$$

Podemos **concatenar** los símbolos *a* y *b* para generar **palabras**, por ejemplo *aaabbb* o concatenar palabras como por ejemplo *aa* y *bbbb* para generar *aabbbb*. El número de símbolos que contiene una palabra se representa cómo |w|. Un punto importante es:

$$|\lambda| = 0$$

El conjunto de todas las palabras generadas por el alfabeto incluyendo la cadena vacía es denotado por:

$$\Sigma^*$$

Que en nuestro ejemplo sería:

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, \dots\}$$

Un lenguaje es conjunto de palabras que pertenecen a Σ^* y que cumplen unas condiciones. Por ejemplo, un lenguaje podría ser aquellas palabras que comienzan por símbolos *a* y terminan por símbolos *b*:

$$L = \{ab, aab, abb, aaab, aabb, abbb, aaaab, \dots\}$$

Este lenguaje contiene infinitas palabras y no se puede escribir todas las secuencias, por lo que no es posible definirlo por extensión, es decir, escribiendo todas las palabras que pertenecen a ese lenguaje, así que se va a definir por comprensión:

$$L = \{a^n b^m : n, m > 0 \ \wedge \ a, b \in \Sigma\}$$

Es común definir los lenguajes formales de esta forma cuando contienen infinitas cadenas, aunque también se puede definir lenguajes con un número de cadenas finitas por comprensión.

4.2. Autómatas finitos deterministas

Un **autómata** finito determinista es una máquina abstracta definida mediante un conjunto de **estados** y un conjunto de **transiciones**, que son una función que dado un estado del conjunto de estados y un símbolo de un alfabeto devuelve otro estado.

Formalmente, un autómata finito determinista (AFD de ahora en adelante) es una 5-tupla tal que:

$$A = (Q, \Sigma, \delta, q_0, F)$$

Siendo Q un conjunto finito de estados, Σ un alfabeto, δ un conjunto finito de transiciones o función de transición, q_0 un estado inicial perteneciente a Q y F un conjunto de estados finales pertenecientes a Q.

Una transición de δ se define como una 3-tupla en la que los dos primeros elementos son el estado de origen de la transición y el símbolo de la transición respectivamente y el tercer elemento el estado destino. De esta forma:

$$\delta(q, s) = q' \iff (q, s, q') \in \delta : q, q' \in Q \ \wedge \ s \in \Sigma$$

Un AFD tiene un lenguaje L asociado que se representa como L(A) y es llamado lenguaje aceptado por el autómata. Este lenguaje es el conjunto de palabras reconocidas por el AFD:

$$L(A) = \{x : x \ \in \Sigma^* \wedge \delta(q_0, x) \to q_1, \ldots, \delta(q_{n-1}, x) \to q_n \wedge q_n \in F\}$$

Recuperando el ejemplo del punto de lenguajes formales, el autómata asociado al lenguaje se define cómo:

$$A = (Q, \Sigma, \delta, q_0, F)$$

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{a, b\}$$

$$\delta = \{(q_0, a, q_1), (q_0, b, q_3), (q_1, a, q_1), (q_1, b, q_2), (q_2, a, q_3), (q_2, b, q_2), (q_3, a, q_3), (q_3, b, q_3)\}$$

$$F = \{q_2\}$$

Con la definición hecha del autómata se puede asegurar que:

$$L(A) = \{a^n b^m : n, m > 0 \ \wedge \ a, b \in \Sigma\}$$

Poniendo un ejemplo con el lenguaje definido, si tenemos una palabra *aaabbb* y queremos comprobar si pertenece al lenguaje, se ha de comprobar si puede ser analizada por el autómata, es decir, el último estado después del análisis debe ser un estado final.

Para esto, se va eliminando o consumiendo el primer símbolo de la cadena en cada transición hasta que no quede ningún símbolo, el estado que quede determinará si la caden pertenece al lenguaje o no:

$$\begin{cases} q_0 & aaabbb \implies \delta(q_0, a) = q_1 \\ q_1 & aabbb \implies \delta(q_1, a) = q_1 \\ q_1 & abbb \implies \delta(q_1, a) = q_1 \\ q_1 & bbb \implies \delta(q_1, b) = q_2 \\ q_2 & bb \implies \delta(q_2, b) = q_2 \\ q_2 & b \implies \delta(q_2, b) = q_2 \\ q_2 & q_2 \in F \end{cases}$$

Como q_2 pertenece a los estados finales, la cadena *aaabbb* pertenece al lenguaje. Esta serie de transiciones para aceptar una cadena serán importantes en el capítulo de inferencia gramatical.

4.3. Gramáticas

Una gramática es un constructo matemático definido por un conjunto de **producciones** formadas por símbolos **terminales** y **no terminales** con un **símbolo inicial**. Mediante las producciones, se puede transicionar de un símbolo no terminal a una cadena de símbolos que pueden ser terminales o no terminales.

Formalmente una gramática es una 4-tupla:

$$G = (V, T, P, S)$$

Siendo V un alfabeto de símbolos no terminales, T un alfabeto de símbolos terminales disjunto con V ($V \cap T = \emptyset$), P una serie de producciones y S un símbolo no terminal llamado axioma de la gramática.

Una producción *p* de P se representa cómo:

$$p = \alpha \to \beta : \alpha, \beta \in (V \cup N)^* \wedge p \ \in P$$

Donde α y β son una cadena de símbolos terminales y no terminales. Dependiendo de las formas de las producciones las gramáticas se pueden clasificar en diferentes tipos, pero las gramáticas relacionadas con este proyecto son las llamadas *lineales pares y lineales.*

Las producciones de una gramática *lineal* tienen la siguiente forma (J. Hopcroft, 1979):

$$A \rightarrow uBv \vee A \rightarrow w : A, B \in V \wedge u, v, w \in T^*$$

Las producciones de una gramática *lineal par* son iguales que las de las gramáticas *lineal* con la diferencia en que las longitudes de *u* y *v* tienen que ser iguales:

$$A \rightarrow uBv \vee A \rightarrow w : A, B \in V \wedge u, v, w \in T^* \wedge |u| = |v|$$

Para dar un ejemplo de esto, se va a usar el lenguaje definido en el apartado de lenguajes formales, que está formado por palabras que comienzan por símbolos *a* y terminan por símbolos *b*:

$$G = (V, T, P, S)$$

$$V = \{S, B, A\}$$

$$T = \{a, b\}$$

$$P = \{S \rightarrow Bb, B \rightarrow Bb, B \rightarrow b, A \rightarrow Aa, A \rightarrow a\}$$

Para formar la palabra *aab* que pertenece al lenguaje L, la secuencia de producciones sería la siguiente:

$$\begin{cases} S \rightarrow Bb & Bb \\ B \rightarrow Aa & Aab \\ A \rightarrow a & aab \end{cases}$$

Así es cómo una gramática genera palabras usando las producciones. Las gramáticas tienen asociados lenguajes formales, que contienen todas las cadenas que pueden ser generadas por la gramática que está relacionada con el lenguaje correspondiente. En el ejemplo dado, la gramática aceptaría el lenguaje definido en el punto 4.1. Con esto se puede ver también la relación entre gramáticas y los autómatas definidos.

4.4. Definición Inferencia gramatical

Con lo explicado en los puntos 4.1, 4.2 y 4.3 se han sentado una base para exponer lo que es la inferencia gramatical y el algoritmo que será utilizado en el actual proyecto.

Recuperando la definición de (de la Higuera, 2010, 27), “La inferencia gramatical es una tarea donde el objetivo es aprender o inferir una gramática para un lenguaje y todo tipo de información de este lenguaje”.

Una de las características de este tipo de aprendizaje automático radica en que se aprende la naturaleza y la estructura de los datos proporcionados para el aprendizaje (Sempere Luna & Heinz, 2016, vii), en algunos casos el autómata o la gramática subyacente a los datos.

Este tipo de aprendizaje usa diferentes tipos de algoritmos para realizar el aprendizaje de las estructuras de los lenguajes formales, uno de los cuales es llamado *algoritmo de identificación de lenguajes k-EE*, que se basa en los conceptos de los lenguajes k-explorables en sentido estricto y que será usado en el presente proyecto.

5. Lenguajes k-explorables y su aprendizaje

El presente proyecto utiliza un algoritmo basado en lenguajes k-explorables en sentido estricto para poder generar un autómata finito determinista a partir de un conjunto de muestras (P. García, 1990), así que se va a proceder a explicar los conceptos subyacentes a esto.

5.1. Lenguajes k-explorables

Un vector de k-explorabilidad de una cadena *x* se puede definir según (P. García, 1990) una 3-tupla:

$$v_k(x) = (i_k(x), t_k(x), f_k(x))$$

Tal que:

$$k > 0 \ \wedge \ x \ \in \Sigma^\star$$

$$i_k(x) = \begin{cases} x & \text{si } |x| < k \\ u : x = uv, |u| = k-1 & \text{si } |x| \geq k \end{cases}$$

$$f_k(x) = \begin{cases} x & \text{si } |x| < k \\ u : x = uv, |v| = k-1 & \text{si } |x| \geq k \end{cases}$$

$$t_k(x) = \{v : x = uvw, u \ \in \Sigma^\star, w \ \in \Sigma^\star, |v| = k\}$$

Por ejemplo, la cadena *aaabbb* con k = 2 tendría vector 2-explorable:

$$v_2(aaabbb) = (a, \{aa, ab, bb\}, b)$$

Una vez definido esto es necesario definir la relación de equivalencia entre cadenas en el campo de la k-explorabilidad cómo:

$$x \equiv_k y \iff v_k(x) = v_k(y) : \forall x, y \ \in \Sigma^*$$

Con todo esto se puede definir un Lenguaje k-explorable, denominado como $k - \mathcal{LT}$, como la unión de clases de equivalencia $\equiv_k$ (García , 1990).

La clase de lenguajes que es utilizada en el proyecto se compone de los lenguajes k-explorables en sentido estricto, denominada como $k - \mathcal{LTSS}$, pues existe un algoritmo de inferencia gramatical que usa los conceptos de esa clase de lenguajes para generar un autómata a partir de una serie de cadenas (García, 1990)

Para dar una definición de lenguajes k-explorables en sentido estricto, dado un lenguaje L, se dirá que es $k - \mathcal{LTSS}$ si:

$$L \cap \Sigma^{k-1}\Sigma^* = (I_k\Sigma^*) \cap (\Sigma^* F_k) - (\Sigma^* T_k \Sigma^*) : I_k, F_k \subseteq \Sigma^{\leq k-1} \ \wedge \ T_k \ \subseteq \Sigma^k$$

Esto es, si las cadenas del lenguaje están formadas por prefijos de I_k, por sufijos de F_k y no contiene segmentos de T_k (García, 1990).

5.2. Aprendizaje de $k - \mathcal{LTSS}$

Mediante los conceptos definidos en el capítulo 5.1 se puede definir un algoritmo en el que, a partir de una serie de muestras que son cadenas de símbolos se infiere un autómata finito determinista que define un lenguaje de la clase $k - \mathcal{LTSS}$.

El algoritmo en pseudocódigo se puede definir de la siguiente manera:

```
Entrada: Un conjunto S de muestras positivas de aprendizaje
Salida: Un AFD AS = (Q, Σ, δ, q0, QF) que reconoce el menor lenguaje k-EE L
que contiene a S
Método:
        Σ = Σ(S); I = Ik(S); T = Tk(S); F = Fk(S); Q = {λ}; δ = ∅; q0 = λ;
        /* Para cada cadena del conjunto de prefijos */
        Para todo a1a2 ... am ∈ I
                δ = δ ∪ {{λ, a1, a1}}
                Para j=1 hasta m
                        Q = Q ∪ { a1a2...aj};
                        δ = δ ∪ {{a1a2...aj-1, aj, a1a2...aj }}
                finPara
        finPara
        /* Para cada cadena del conjunto de infijos */
        Para todo a1a2 ... ak ∈ T
                Q = Q ∪ { a1...ak-1, a2...ak};
                δ = δ ∪ {{a1...ak-1, ak, a2...ak }}
        finPara
        /* Los estados finales son los sufijos */
        QF = F;
        AS = (Q, Σ, δ, q0, QF )
finMétodo
```

Algoritmo que infiere lenguajes k-explorables

En este método, el parámetro del que se habló en el capítulo 3.2 sería el valor de k, que dependiendo de él se generará un autómata o otro. Sabiendo esto, una parte importante de la fase de experimentación será variar el parámetro k para decidir cuál es el valor más aconsejable.

El algoritmo se aprovecha de la definición de los lenguajes k-explorables en sentido estricto para generar un autómata que acepte las cadenas pertenecientes a las muestras. Por eso se obtienen los prefijos, sufijos e infijos, I_k, F_k y T_k respectivamente, y se va construyendo el autómata según estos conjuntos.

6. Autómatas de Watson-Crick

Antes de pasar al capítulo de estado del arte y de la solución propuesta es necesario definir lo que es un autómata de Watson-Crick y sus diferentes tipos, así como la relación con los lenguajes formales, porque será utilizado en el momento de anotar una secuencia de ADN en la solución propuesta.

Es necesario mencionar que siempre que se haga referencia a un autómata de Watson-Crick se estará hablando realmente de un autómata **finito** de Watson-Crick, pero se obvia el término finito por simpleza.

6.1. Definición formal

Primeramente es necesario definir el concepto de reverso de una cadena, que es simplemente leer una cadena al revés y será representado como x^r. Por ejemplo, el reverso de la cadena *aaabbb* sería:

$$aaabbb^r = bbbaaa$$

Una vez explicado esto, se va a introducir el concepto de molécula definido en (Paun et al.) y se va a relacionar con los autómatas de Watson-Crick.

6.1.1. Molécula

Dado un alfabeto Σ, un *sticker* es un par de cadenas tal que:

$$(x = x_1 v x_2, y = y_1 w y_2) : x, y \in \Sigma^* \ \wedge \ p(v) = w$$

Siendo p una función que relaciona una cadena con otra cadena llamada cadena complementaria. Un *sticker* puede ser denotado como:

$$\binom{x}{y}$$

Una molécula completa y complementaria es un sticker tal que:

$$|x| = |y| \text{ y } p(x) = y$$

y será denotada como:

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

Cualquier molécula o *sticker* puede ser denotado como:

$$x \# y^r : \# \notin \Sigma$$

En la solución adoptada se cambiará el símbolo # por el símbolo -.

Poniendo un ejemplo de esto, supongamos que tenemos una cadena *aaabbb* cuya cadena complementaria es *cccddd*, una molécula de estas cadenas sería escrita como:

$$\begin{bmatrix} aaabbb \\ cccddd \end{bmatrix} = aaabbb\#dddccc$$

6.1.2. Autómata de Watson-Crick

Un autómata de Watson-Crick, de ahora en adelante AFWK es definido como una 6-tupla:

$$M = (V, \rho, K, s_0, F, \delta)$$

Siendo V y K un conjunto de símbolos disjuntos (que representan los símbolos de las cadenas y los estados del autómata), ρ una relación simétrica entre elementos de V, s_0 un símbolo de K (que representa el estado inicial), F un subconjunto de K (que representa el conjunto de estados finales) y δ una función de transición definida de la siguiente forma:

$$\delta : K \times \binom{V^*}{V^*} \to P(K) : \delta\left(s, \binom{x}{y}\right) \neq \emptyset \wedge (s, x, y) \in K \times V^* \times V^*$$

donde P(K) representa el conjunto de las partes de K, es decir todos los subconjuntos de K.

De forma similar a los AFD, el autómata analiza una cadena de símbolos, pero esta vez en vez de ser una simple cadena como en el caso del AFD, lo que analiza el AFWK es una molécula, definida en el punto 6.1.1.

A continuación se va a exponer un ejemplo que va a ser el mismo dado en el punto 4.2 pero cambiando los símbolos de las transiciones por moléculas. Supongamos que tenemos un AFWK definido de la siguiente forma:

$$V = \{a, b, c, d\}$$

$$\rho(x) = \begin{cases} c & x = a \\ d & x = b \end{cases}$$

$$K = \{v_0, v_1, v_2, v_3\}$$

$$s_0 = v_0$$

$$F = \{v_2\}$$

Las transiciones de δ se van a mostrar de modo gráfico en lugar de lista, para que sea más explicativo:

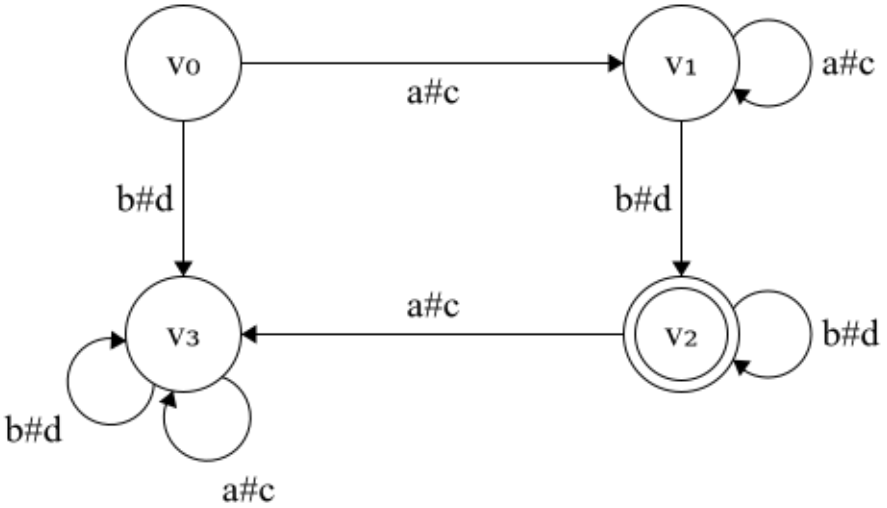

Imagen 3: Autómata de Watson-Crick ejemplo gráfico de transiciones

Ahora supongamos que queremos analizar la molécula:

$$\begin{bmatrix} aaabbb \\ cccddd \end{bmatrix}$$

La transición de estados permite ir eliminando dos símbolos en cada paso, uno por el inicio y otro por el final, por eso el apunte de que la segunda parte de la molécula escrita con la forma $x\#y^r$ tenga la segunda parte como reverso. Cuando sólo quede el símbolo # de la cadena o la molecula quede vacía, se comprobará si la molécula es aceptada por el AFWK cerciorando si el estado que queda es final o no:

$$\begin{cases} v_0 \begin{pmatrix} aaabbb \\ cccddd \end{pmatrix} \implies \delta\left(v_0, \begin{pmatrix} aaabbb \\ cccddd \end{pmatrix}\right) = v_1 \\ v_1 \begin{pmatrix} aabbb \\ ccddd \end{pmatrix} \implies \delta\left(v_1, \begin{pmatrix} aabbb \\ ccddd \end{pmatrix}\right) = v_1 \\ v_1 \begin{pmatrix} abbb \\ cddd \end{pmatrix} \implies \delta\left(v_1, \begin{pmatrix} abbb \\ cddd \end{pmatrix}\right) = v_1 \\ v_1 \begin{pmatrix} bbb \\ ddd \end{pmatrix} \implies \delta\left(v_1, \begin{pmatrix} bbb \\ ddd \end{pmatrix}\right) = v_2 \\ v_2 \begin{pmatrix} bb \\ dd \end{pmatrix} \implies \delta\left(v_2, \begin{pmatrix} bb \\ dd \end{pmatrix}\right) = v_2 \\ v_2 \begin{pmatrix} b \\ d \end{pmatrix} \implies \delta\left(v_2, \begin{pmatrix} b \\ d \end{pmatrix}\right) = v_2 \\ v_2 \implies v_2 \in F \end{cases}$$

Como al final se llegó a un estado que es final, la molécula es aceptada por el AFWK.

6.2. Tipos de AFWK

Las transiciones de un AFWK no sólo están limitadas a un estado y una molécula, sino que pueden tener cualquier tipo de *sticker*. Por eso, según el tipo de transición que tenga un AFWK se puede realizar la siguiente clasificación (Păun et al., 1998, 154):

- *stateless*: En este tipo de autómata el estado inicial es el único estado que contiene el AFWK, por lo que también es el único estado final.

$$K = F = \{v_0\}$$

- *all-final*: Este AFWK es aquel en el que todos sus estados son finales.

$$K = F$$

- *simple*: En este caso, en todos los *stickers* de las transiciones, una de las dos cadenas es la cadena vacía:

$$\forall v_i, v_i' \in K,\ \delta\left(v_i, \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = v_i' : x_1 = \lambda \oplus x_2 = \lambda$$

- *1-limited*: Este tipo de AFWK es parecido al *simple*, con la condición de que la concatenación de las dos componentes del *sticker* tienen que tener longitud 1:

$$\forall v_i, v_i' \in K,\ \delta\left(v_i, \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = v_i' : |x_1 x_2| = 1$$

Estos tipos de autómatas pueden ser mezclados para generar AFWK compuestos, por ejemplo, se podría generar un autómata *1-limited* y *all-final*, aunque esta explicación se sale del alcance de este proyecto, pues el tipo de autómata en el que se interesa este proyecto es el tipo *1-limited*, que será usado en el siguiente punto, el 6.3.

6.3. Relación con los lenguajes formales

En el punto 6.1.2 se ha podido ver una especie de similitud entre los AFD y los AFWK, aunque no se ha mencionado formalmente esto hasta el momento. En este punto se va a explicar la relación entre los lenguajes formales, los AFD y los lenguajes k-explorables con los AFWK. Las explicaciones de este punto serán obtenidas de (Sempere, 2007).

Primeramente se ha de mencionar que el lenguaje de las moléculas aceptado por un AFWK *M* será denotado como $L_m(M)$. También, con un AFWK se puede definir un lenguaje llamado $L_u(M)$, que es el lenguaje cuyas cadenas son la parte superior de las moléculas aceptadas por un AFWK *M*.

Como se puede intuir, $L_u(M)$ está relacionado con $L_m(M)$, pero obteniendo sólo la primera parte de una molécula (la parte *x* de *x#y*').

Según (Sempere, 2007) se puede demostrar que:

$$L_m(M) = L_1 \cap L_2$$

Siendo L_1 un lenguaje *lineal* y L_2 un lenguaje *lineal* definidos en el punto 4.3. La demostración de este teorema excede el alcance de este proyecto, pero se puede encontrar perfectamente definida en (Sempere, 2007).

Como se ha visto en el punto 6.1.2, un AFWK es muy parecido a un AFD, y se puede ver fácilmente, tal y como se establece en (Sempere, 2007), que se puede obtener un AFD de un AFWK si sólo se analiza la parte superior (o primera) de una molécula. También se puede obtener un AFD con la parte inferior de una molécula, así que se podrían obtener dos AFD a partir de un AWC:

- AFD_u: El AFD definido por la parte superior de las moléculas de un AFWK
- AFD_l: El AFD definido por la parte inferior de las moléculas de un AFWK

Recuperando el ejemplo del punto 6.1.2, se podría dividir el AWC en dos diferentes, uno para cada parte de las moléculas:

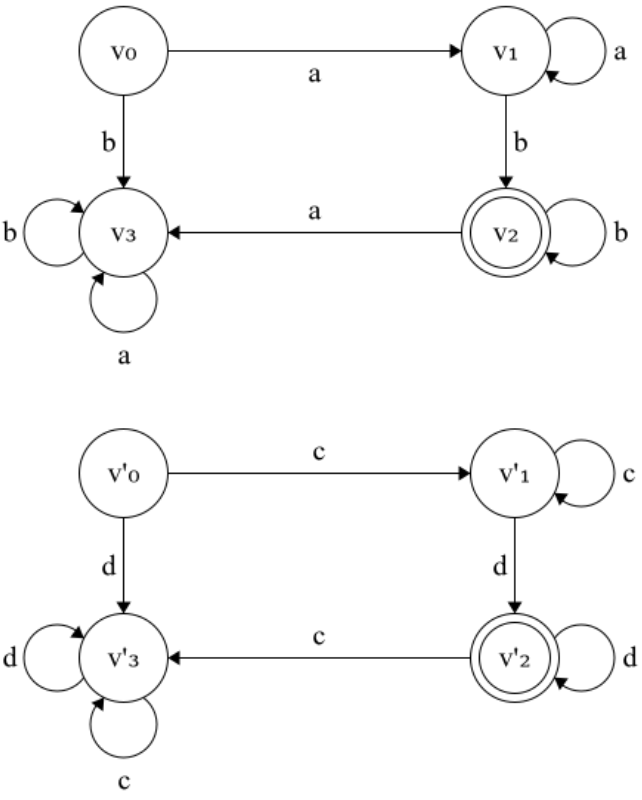

Imagen 4: AFD_u y AFD_l generados desde un AWC

Con esto se puede ver, como se dijo en el capítulo 4.2, que un AFD tiene un lenguaje formal asociado. De esta forma se podría relacionar el AFD de la parte superior de una molécula con los lenguajes k-explorables y k-explorables en sentido estricto.

Se podría generar un lenguaje k-explorable en sentido estricto para un AFD_u y asociar un AFD_l que se pueda generar a partir de una función de complementariedad a partir del alfabeto Σ del AFD_u, y a continuación transformar estos dos AFD en un AFWK.

Un proceso parecido al anterior será el usado para anotar secuencias de ADN usando un AFWK, que será generado a partir del AFD generado en una fase de entrenamiento de un modelo de lenguajes k-explorables en sentido estricto.

7. Propuesta de una solución para el anotador genómico

El uso de métodos de aprendizaje automático cada vez abarca más campos de la vida de las personas. Aplicaciones como *Instagram* o *Snapchat* utilizan estos modelos para poder aplicar sus filtros, o para saber qué y cuándo sugerir *posts* a sus usuarios (Intel, n.d.).

Más allá del apartado social y lúdico de la sociedad, también se usa en ámbitos empresariales y académicos, cómo en la gestión del contenido *spam*, en la extracción de analíticas en grandes cantidades de datos o en saber las opiniones de sus usuarios en redes sociales con análisis del lenguaje natural (Velogig, 2020). Así que para la realización del presente proyecto se planteó si se podría usar conceptos cómo estos en el ámbito de la medicina, específicamente en el campo de la genética.

Buscando información sobre ello, se pudo ver que actualmente hay varios estudios relacionados con el aprendizaje automático y el ámbito de la genética, por ejemplo, en (Jamal et al., 2020) se usaron técnicas de aprendizaje automático para predecir mutaciones que pueden causar resistencia ante medicamentos respecto a la enfermedad de la Tuberculosis. Estos modelos dieron muy buenos resultados, todos con un acierto mayor al 70%.

En el ejemplo anterior no se usaron técnicas de inferencia gramatical, sino otras cómo *deep learning*, *SVM* o vecinos más cercanos. Así que en este proyecto se planteó la posibilidad de aplicar la inferencia gramatical en la predicción de mutaciones genéticas usando un algoritmo basado en los lenguajes k-explorables en sentido estricto y autómatas de Watson-Crick, pues hasta el momento no se había probado en ningún proyecto previo de la ETSINF en este ámbito.

El uso de autómatas de Watson-Crick fue elegido por su similitud con el ADN. Una forma en la que se explica es en (Nagy et al., n.d.), que se puede ver la relación entre lo conocido cómo *DNA computing* y los AFWK, por lo que se buscaría una forma de relacionar el AFD generado por el algoritmo de los lenguajes k-explorables con un AFWK.

Con todo esto se pretende ver la capacidad de aprendizaje de este algoritmo aplicado a las mutaciones genéticas y sacar conclusiones de si es un modelo adecuado, de si se debe ampliar el modelo, añadiendo una capa estocástica por ejemplo, o si la aplicación de este algoritmo no es el adecuado para este ámbito o caso de estudio en específico.

Para ello, se van a usar las herramientas que se disponen en la actualidad y que están más extendidas en el campo del aprendizaje automático, del desarrollo de software y del tratamiento de datos.

Respecto a las tecnologías utilizadas, el lenguaje de programación elegido es python, debido a la facilidad que presenta a al hora de realizar programas, así como la cantidad de librerías *open source* que permiten no hacer código que ya esté implementado por otro usuario. También se ha decidido usar este lenguaje porque es uno de los lenguajes más usados en el ámbito del aprendizaje automático y el *data science* (Morales, n.d.).

Para la gestión y evolución del presente proyecto se utilizará las metodologías TDD y SCRUM así como el seguimiento de las directrices dictadas en las *best practices* de *Python*, pues uno de los problemas en los desarrollos de soluciones de aprendizaje automático que existen en los repositorios son la falta de facilidad para la legibilidad y la dificultad para la realización de cambios en librerías ya existentes.

Antes de seguir, es necesario mencionar que al adoptar estas metodologías de trabajo, el desarrollo pasaría a costar más tiempo, pues el seguimiento de estas lleva tiempo de análisis y gestión adicional, pero se podría garantizar que ante cambios la solución planteada se podría adaptar de forma más sencilla.

Así que uno de los principales objetivos de este proyecto es la ejecución y comprobación del algoritmo de los lenguajes k-explorables y el desarrollo de una librería escalable y fácilmente manipulable, tanto por usuarios como por desarrolladores.

7.1. Caso de estudio

En el presente proyecto se va a tratar la enfermedad conocida por el nombre de *distrofia de retina*. Según (Miranza, n.d.) “es una enfermedad hereditaria de la retina, que afecta a sus capas más externas, dañando principalmente las células fotorreceptoras (un tipo de neuronas sensibles a la luz) y el epitelio pigmentario (encargado de nutrir a los fotorreceptores). En algunas distrofias, también pueden producirse alteraciones en el gel vítreo que rellena el globo ocular y está en contacto con la retina”.

Esta enfermedad tiene una causa genética, por eso se ha elegido para este proyecto. Las mutaciones provocan que las células de la retina funcionen de manera incorrecta, conduciendo a su degeneración.

Así que se va a tratar de obtener una serie de secuencias de ADN mutadas relacionadas con la enfermedad para que, una vez obtenidas, poder entrenar el modelo de lenguaje k-explorable y que a partir de un AFWKC y secuencias nuevas se pueda decir dónde y qué tipo de mutación se produce.

Se ha decidido usar el camino de la inferencia gramatical para diversificar en el ámbito del aprendizaje automático y comprobar cómo se desenvuelven los algoritmos de inferencia gramatical en el ámbito de la genética. A pesar de que otros modelos dan buenos resultados, como en el caso del *deep learning*, se quiere comprobar cómo se comportan otros tipos de algoritmos en este campo del conocimiento.

Ya se ha visto en estudios previos que la inferencia gramatical es útil en el apartado de la genética (poner la cita de Sakakibara), por eso se plantea usar el método de los lenguajes k-explorables y usar un AFWKC para poder gestionar las predicciones usando el modelo generado creando anotaciones a partir de secuencias de ADN.

Una vez realizada la implementación y la comprobación de los resultados se podrán sacar conclusiones sobre si el método es el adecuado o no.

7.2. Solución propuesta

La solución adoptada pasará por tres fases que son típicas en los desarrollos relacionados con el aprendizaje automático:

- Preparación del proyecto: La primera fase consiste en preparar clases útiles para la gestión de argumentos de consola y la muestra de datos por pantalla, así cómo la configuración del proyecto en el repositorio correspondiente.

- Obtención de datos: En esta fase se diseñarán e implementarán las clases de *Python* relacionadas con la extracción de datos desde un fichero FASTA haciendo uso de las mutaciones presentes en ficheros VCF.

- Transformación de datos: En esta fase se diseñarán e implementarán clases que transformarán las muestras obtenidas por la fase anterior y serán preparadas para poder ser usadas por el método de inferencia gramatical.

- Generación del modelo: En esta fase se diseñarán las clases que generarán el modelo para que pueda ser probado en la siguiente fase.

- Prueba y extracción de conclusiones: En esta fase se probará el modelo diseñando clases que permitan obtener las métricas de efectividad del modelo y se obtendrán las conclusiones y el trabajo futuro.

Para afrontar esta tarea se ha decidido diseñar la solución como una librería para el lenguaje de programación *Python*. De esta forma resultaría más fácil poder extenderlo para trabajos futuros.

Aunque *R* también es un lenguaje bastante extendido en el campo, se decidió usar *Python* por la cantidad de librerías relacionadas con el ámbito de la genética y el tratamiento de textos.

Para el desarrollo del proyecto, se ha decidido adoptar la metodología de trabajo TDD, ya que de esta forma, al cambiar código o añadir nuevo código, es más fácil comprobar que todo funciona correctamente ejecutando los *tests*.

Con eso, se podría facilitar la tarea del mantenimiento y la escalabilidad, pues esos conceptos serán importantes porque la metodología de trabajo que se usará se basa en evolutivos, y en cada evolutivo se podrán realizar cambios en lo que ya se haya desarrollado. Por lo que será muy útil tener *tests* que validen que los cambios no afecten a los resultados de lo que ya se haya realizado.

Respecto al desarrollo de la solución, se ha decidido dividir cada funcionalidad en una clase diferente de *Python*, para aislar las funciones y permitir una mayor

facilidad a la hjora de programar los *tests*. En el punto 10 (atención a la enumeración final del índice) se explicará en más detalle las clases usadas y la relación que hay entre ellas, así cómo su utilidad.

A parte de una serie de clases de *Python*, también se desarrollarán clases que permitan una mejor forma de manejar los parámetros que se introducirán por los usuarios mediante el terminal, así cómo la gestión de las estructuras de datos que se usuarań en el proyecto

Siguiendo con el tema de la evolución del proyecto, se utilizará la metodología SCRUM por su facilidad para realizar cambios y comprobar los avances en los proyectos de desarrollo de software. Para apoyar el desarrollo de las funcionalidades en consonancia con SCRUM se utilizará la plataforma Trello para la gestión de las tareas.

Cómo se explicó en el punto 1.4, se desarrollarán una serie de evolutivos para cada fase del proyecto, ajustándose el número de evolutivos por fase dependiendo del avance de cada una en el momento de planificar cada evolutivo.

Un evolutivo durará dos semanas, en las cuales se planificará una serie de desarrollos que serán estimados y que se intentarán hacer para el siguiente evolutivo. Si no se consigue acabar alguna funcionalidad en un evolutivo, esa funcionalidad será desplazada al siguiente evolutivo y será vuelta a estimar junto con otras funcionalidades a implementar en el nuevo evolutivo.

8. Herramientas utilizadas

En este capítulo se va a explicar brevemente en qué consisten las herramientas utilizadas y porque se han decidido utilizar para el proyecto.

8.1. Ubuntu

Para el desarrollo del proyecto se ha decidido utilizar Ubuntu como sistema operativo, pues es un sistema basado en Debian, que a su vez está basado en linux. El sistema operativo es *open source* y el hecho que esté basado en linux es muy beneficioso, ya que los sistemas linux son ampliamente utilizados por la comunidad de desarrolladores de software.

Así que se ha elegido este sistema por varias razones que se van a exponer a continuación:

- Muchas de las librerías de *Python* están desarrolladas para sistemas linux, así que una opción correcta es desarrollar el proyecto en un sistema operativo con base en linux.

- Si los resultados del proyecto son satisfactorios se puede desarrollar un API, por ejemplo REST, y montarla en un servidor para que otros usuarios puedan utilizar las funcionalidades del proyecto de cara a un desarrollo futuro.

- Ubuntu presenta un entorno de escritorio muy amigable con el usuario, parecido a Windows, pero con las ventajas de los sistemas linux.

- La facilidad del manejo del terminal y las instrucciones que contiene, así como la compatibilidad con archivos de tipo *Makefile* hacen que Ubuntu sea un sistema operativo con las funcionalidades requeridas por el proyecto.

8.2. Python

Python es un lenguaje muy usado, especialmente en el ámbito de la inteligencia artificial y del aprendizaje automático, por lo que posee mucha documentación y funcionalidades. Además, este lenguaje posee grandes cantidades de librerías útiles para el tratamiento de datos y de texto.

Una de las ventajas de este lenguaje es la facilidad en la que se puede programar, pues las sentencias e instrucciones son muy amigables respecto a usuarios que no sean profesionales en el ámbito del desarrollo de software. También permite una fácil gestión de los paquetes creados de las librerías.

Además, *Python* posee una herramienta de autogeneración de código llamada *pdoc*, que será utilizada para crear una documentación del código desarrollado en

formato *HTML*. Gracias a esto si el proyecto resulta exitoso se podría subir esta documentación a un servidor para que pudiera ser consultada por cualquier usuario.

Es necesario mencionar que para la gestión de librerías externas se ha usado la herramienta llamada *poetry*, que es muy útil a la hora de manejar las librerías e instalarlas o actualizarlas.

Finalmente, se han usado muchas librerías relacionadas con *Python* para la implementación en las clases en las que se vayan a utilizar sus funcionalidades, pero se comentarán en el capítulo 10 de implementación en las clases en las que se vayan a utilizar sus funcionalidades.

8.3. Makefile

Antes de continuar con la explicación, se asume que el lector tiene conocimientos básicos de tratamiento de instrucciones usando un terminal o consola de comandos.

Para la gestión de los parámetros que se deberán pasar por consola de comandos se ha decidido utilizar un tipo de archivo llamado Makefile, que es una utilidad de distribuciones GNU-linux.

Este tipo de archivos permiten definir una serie de parámetros de línea de comando y abreviarlos en un solo comando.

Supongamos que queremos filtrar las cadenas de un archivo que empiezan por el carácter '>' y guardar el resultado en un archivo llamado *result.txt*, pues podríamos definir un archivo Makefile que contuviera la siguiente información:

```
save:

    cat file.txt | grep '>' > result.txt
```

Para ejecutar el comando, sólo tendríamos que ejecutar la instrucción en el mismo directorio que estuviera el archivo makefile:

```
make save
```

Este fichero contendrá una forma simplificada de ejecutar la clase *Runner*, que se encarga de ejecutar todo el proceso para generar el modelo.

8.4. Visual Studio Code

Para el desarrollo de código se ha decidido usar el IDE de programación Visual Studio Code, VSC de ahora en adelante, pues proporciona muchas funcionalidades para el desarrollo de software con *Python*, cómo gestión de errores de sintaxis, estructuración de código automática o sugerencia de métodos y variables.

Otra de las razones por las que se decidió utilizar VSC es la integración con el sistema de gestión de versiones github y la integración con la herramienta pytest para la gestión, ejecución y depuración de los tests generados mediante la metodología TDD.

Para poder seguir con la siguiente explicación es necesario explicar el concepto de punto de parada, y significa que cuando se quiere depurar un código, es decir, tratar de eliminar errores, se puede especificar en algunos IDEs una instrucción en la que el IDE puede pararse para que el desarrollador pueda consultar las variables en el momento de la parada.

Siguiendo con las razones por las que se eligió VSC, una de las ventajas por las que se decidió usar el IDE es la gran capacidad de configuración a la hora de depurar código, pues la forma en que se integran los puntos de parada y el manejo de las variables locales cuando se está en un punto de parada de un método es realmente útil en caso de que hayan errores en la ejecución y se quiera investigar la causa de estos errores.

8.5. Pytest

Respecto al desarrollo de tests mediante TDD, se decidió usar pytest tanto como librería como gestor de tests. Pytest es un gestor de tests que proporciona muchas funcionalidades en relación con test unitarios y descubrimientos de tests automáticos en un directorio o librería de *Python*.

Esto es debido a la integración y la configuración que permite VSC en relación con pytest. Por las funcionalidades que implementa como librería se decidió usar pytest como gestor de tests.

8.6. Github

Primeramente hay que definir el concepto de git. Git es un sistema que se encarga de gestionar las diferentes versiones de un código. Gracias a este sistema, cada nueva funcionalidad se puede desarrollar en un entorno diferente del que se tiene desarrollado el código que ya esté realizado.

Esto es gracias al concepto de rama, pues cada rama representaría una fase diferente del desarrollo de un proyecto, fases que en un momento dado, se podrían fusionar y mezclar el código generado en ambas fases. Este concepto será importante en las explicaciones siguientes, específicamente en el punto en el que se habla de *Pull Requests*.

Github es una plataforma online para la gestión de versionados de código que será utilizada por el proyecto para mantener el código, gestionar los evolutivos y los cambios que se hagan así cómo para consultar el histórico de cambios realizados en el proyecto.

El método de trabajo con Github será usar la funcionalidad llamada *Pull Request*, que consiste en agrupar un conjunto de cambios de una rama y fusionarla con otra.

Aunque hay diferentes plataformas que permiten gestionar el código online, cómo Gitlab o Bitbucket, se prefirió usar Github por la facilidad que proporciona respecto a gestión de la seguridad, específicamente la parte de claves *ssh*, y la forma en que muestra el código en las *Pull Requests*.

8.7. Trello

Para la gestión de tareas se decidió usar Trello, pues proporciona un ambiente muy simple y una vista de tipo *Kanban* en las que se puede manejar la creación, edición y gestión de las tareas a desarrollar, así como cambiar las tareas dependiendo de la fase en las que se encuentren.

9. Información genómica

Antes de continuar se debe especificar el tipo de ficheros de los que se han obtenido los datos para entrenar el modelo de lenguajes k-explorables. Toda la información de este capítulo ha sido obtenida de (Wing-Kin, 2017).

9.1. FASTA

Un archivo FASTA es un tipo de archivo que contiene toda la secuenciación de genes de una especie, que en el caso del presente proyecto será el genoma humano. En nuestro caso, el archivo FASTA está dividido en cromosomas, en el que cada cromosoma viene introducido por un símbolo '>' y el identificador del cromosoma.

Por ejemplo, un posible fichero FASTA podría tener la siguiente forma:

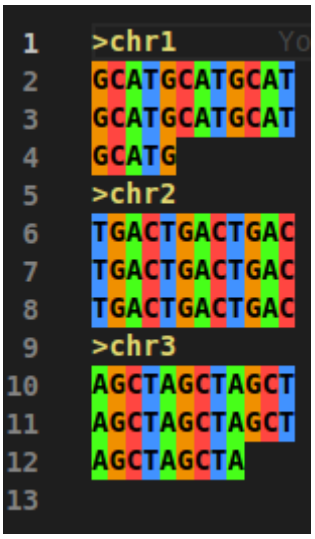

Imagen 5: Fichero FASTA de ejemplo

En este fichero de ejemplo se han caracterizado los nucleótidos con colores para que sea más fácilmente legible. Como se puede apreciar, cada fragmento de secuencia de cada cromosoma viene identificado con el símbolo '>' y su nombre, mientras que a continuación se expone la serie de nucleótidos que componen el fragmento.

9.2. VCF

Un fichero VCF contiene datos sobre mutaciones ocurridas en los genomas de un genoma específico. Esta clase de ficheros contienen muchos datos en relación con conceptos de biología y medicina, conceptos que exceden el alcance del presente proyecto, ya que lo que es interesante es que por cada mutación, se especifica el cromosoma y la posición dentro del cromosoma donde ha ocurrido la mutación.

Un ejemplo de fichero VCF con los campos relacionados con el proyecto sería:

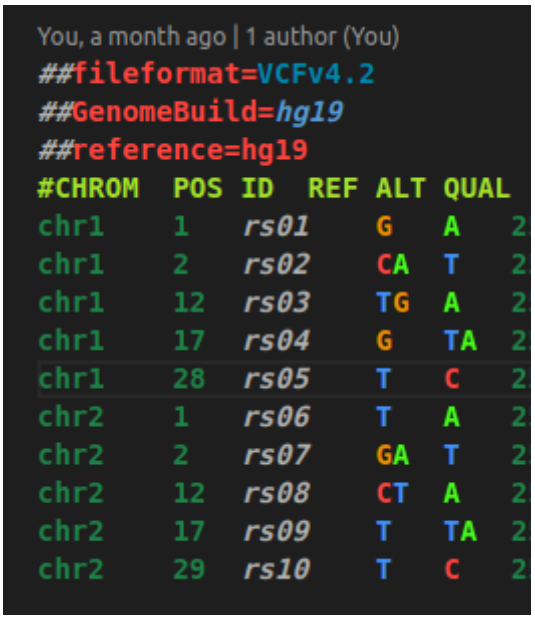

Imagen 6: Ejemplo de fichero VCF

Como se puede apreciar, en el fichero se especifican las mutaciones y las posiciones correspondientes de cada mutación, así como la referencia y la mutación ocurrida y el cromosoma donde ha ocurrido.

10. Implementación

En este capítulo se van a explicar los detalles de la implementación de la solución en código, así como el diseño y la arquitectura usada. Primeramente se va a comentar el diseño pensado para el proceso entero de entrenamiento y validación de un modelo, que consistiría en obtención y preparación de los datos, generación del modelo y validación.

Durante todas las fases se deberá dar retroalimentación al usuario, por lo que mientras se ejecuta cada fase se enviarán una serie de mensajes al terminal usando la herramienta *logger* de *Python* para mostrar el estado actual del proceso. *Logger* es una librería de *Python* que permite emitir mensajes con más información que la sentencia *print*, como la hora, el día o el tipo de mensaje.

Después de la explicación de las fases, se procedería a mostrar en detalle en qué consiste cada paquete de la librería y sus clases, así como el funcionamiento de la clase *Runner*, que permite realizar el proceso completo de obtención de datos, entrenamiento y validación de forma automática.

Finalmente se procedería a mostrar la solución que se ha elegido para el proyecto y su ejecución, después de explicar todas las fases del proceso de generación del modelo y los detalles de cada parte de la implementación de las clases.

10.1. Fases

El siguiente esquema relaciona cada fase con las clases que la componen y acto seguido se procederá a explicar en qué consiste cada fase, que clases principales se usan y cómo se tratan los datos:

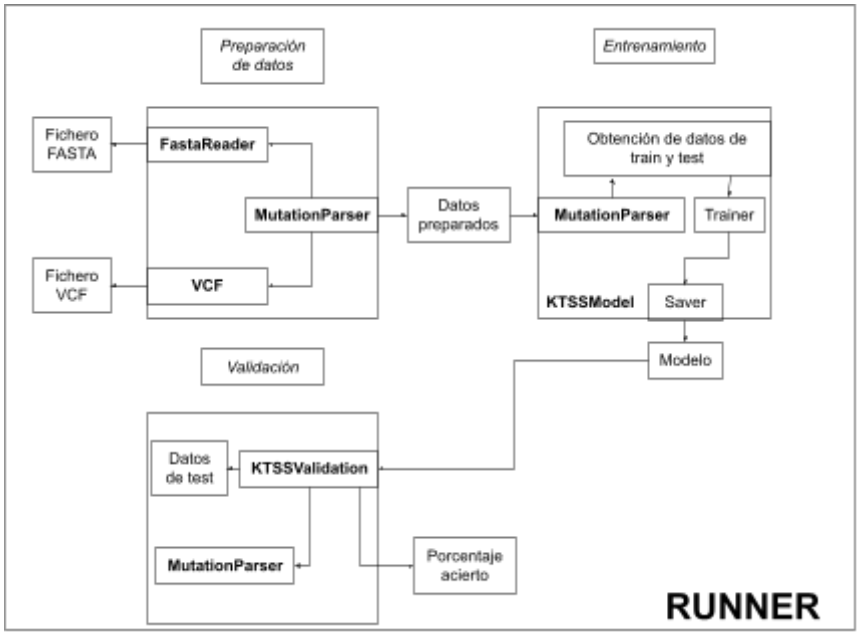

Imagen 6: Diagrama de arquitectura de la librería

10.1.1. Preparación de datos

La primera fase se encarga de recopilar una serie de secuencias que contienen una mutación y transformarlas a otra notación para a continuación insertarlas en un archivo. El hecho de la inserción de datos en un archivo está hecho para que cada fase sea independiente y se puedan realizar de manera asíncrona, que no desordenada, de esta forma se podría montar un *pipeline* para entrenar varios modelos a la vez.

El proyecto se basa en predecir mutaciones, así que lo primero que se debe hacer es obtener una serie de secuencias que contienen mutaciones para poder entrenar el modelo. Para obtener estas mutaciones, se necesitan dos archivos, uno VCF para obtener la información de las mutaciones y un archivo FASTA para obtener secuencias de ADN a partir de las posiciones especificadas en el VCF. Acto seguido se deberían transformar esas secuencias a otra notación para que el modelo pudiera entrenar en base a esa nueva notación.

Así que para esta labor se usarían dos clases, *FastaReader* y las clases que hicieran herencia de *ParserVCF.* Estas clases se encargan de obtener datos del fichero FASTA, obtener secuencias mutadas y transformarlas. Adicionalmente se usaría una clase llamada VCF para poder obtener datos a partir de un fichero VCF de la librería *PyVCF*.

Toda la gestión de obtener datos de los ficheros y transformarlos es gestionada por la clases que hereden de *ParserVCF*, que es una clase abstracta que se puede heredar para definir diferentes tipos de *mappers*, que serán explicados en el capítulo de anotación 10.1.1.2.

En esta fase se pueden observar tres principales pasos para la generación del archivo con las secuencias anotadas con mutaciones, que son el paso de obtención de mutaciones, el paso de anotación y el de escritura en el archivo de mutaciones. Las tres partes se pueden hacer de forma concurrente línea a línea del fichero VCF, por lo que en un futuro trabajo sobre el actual proyecto se podría paralelizar y hacer más eficiente la fase de preparación de datos.

A continuación se va a proceder a explicar en qué consiste cada paso y sus características.

10.1.1.1. Obtención de mutaciones

En esta parte de la primera fase se obtienen las secuencias mediante la clase *VCF*. Lo que se realiza es que para cada mutación incluida en el fichero *VCF* se obtiene la posición, la referencia y la mutación para enviarselo al anotador para que genere una serie de secuencias anotadas para el modelo.

La clase *VCF* es parte de una librería llamada *PyVCF* que proporciona funcionalidades para la gestión de ficheros de tipo VCF y obtener datos a partir de estos, como el número de mutaciones.

Para obtener las secuencias de ADN era necesario obtener el genoma completo relacionado con el fichero *VCF*. Para esto se utilizaría la web *Ensembl*, que permite descargar ficheros relacionados con genómica, como el genoma humano en nuestro caso.

10.1.1.2. Anotación

Respecto a la labor de transformación, que de ahora en adelante se denominará como anotación, consiste en cambiar caracteres de la secuencia por otros. Por ejemplo, supongamos que tenemos la cadena *AAACCC* en la que hay una mutación en el centro, que es *AC*.Podríamos cambiar la notación según si un símbolo es prefijo, mutación o sufijo.

Cada secuencia será representada como una 3-tupla, en la que el primer elemento será el prefijo, el segundo elemento será el infijo o mutación y el último elemento será el sufijo de una secuencia:

$$S = (S_p, S_m, S_s) : S_i \in \{A, C, G, T\} \wedge i \in \{p, m, s\}$$

Así que la cadena *AAACCC* que contiene una mutación *AC* será representada cómo:

$$S = (AA, AC, CC)$$

Para gestionar la anotación en la fase de tratamiento de datos, se pueden definir una serie de asociaciones o *mappings*, que se encargan de asociar un símbolo de la cadena original a otro diferente. En el ejemplo dado, los *mappings* podrían consistir en tres diferentes, uno para el prefijo, uno para la mutación y otro para el sufijo:

$$m_p = \{(A, p), (C, p), (G, p), (T, p)\}$$

$$m_m = \{(A, m), (C, m), (G, m), (T, m)\}$$

$$m_s = \{(A, s), (C, s), (G, s), (T, s)\}$$

$$m_i(s) = s' : i \in \{p, m, s\} \wedge (s, s') \in m_i$$

Aplicar un *mapping* a una secuencia de símbolos es equivalente a aplicar un *mapping* a cada símbolo de la secuencia, es decir:

$$P = P_0 P_1 \dots P_N \rightarrow m_i(P) = m_i(P_0) m_i(P_1) \dots m_i(P_N)$$

Cada *mapping* será aplicado a un elemento de la 3-tupla, asignando los *mappings* de prefijo, mutación y sufijo al prefijo, mutación y sufijo de la tupla respectivamente. Es decir, si quiero anotar la secuencia S, se debería hacer lo siguiente:

$$Map(S) = (m_p(S_p), m_m(S_m), m_s(S_s)):$$

Con las definiciones del ejemplo anterior se puede observar que el prefijo de una cadena será codificado con el símbolo *p*, el infijo o mutación con el símbolo *m* y el sufijo con el símbolo *s*. Así que la cadena *AAACCC* en la que hay una mutación en el centro, que es *AC* quedaría como:

$$S' = (pp, mm, ss)$$

En el presente proyecto se ha usado un anotador en la fase de obtención de datos llamado *MutationParser*, que heredará de *ParserVCF*. Este anotador anota cada símbolo dependiendo de si es sufijo, mutación o prefijo. En el caso del prefijo y el sufijo se anotaría de la siguiente forma:

$$m_p = \{(A, q), (C, w), (G, e), (T, r)\}$$

$$m_s = \{(A, z), (C, x), (G, c), (T, v)\}$$

En el caso del *mapping* de mutaciones, es algo diferente, porque antes de realizar la anotación, se realiza el algoritmo de distancia de Levenshtein entre la cadena mutada y la cadena de referencia, que consiste en ver cuantas operaciones hay que hacer para transformar una secuencia en otra. Por ejemplo, la distancia de Levenshtein entre casa y calle sería de tres, sustituir *s* por *l*, añadir una *l* y sustituir *a* por *e*.

Como se ha visto en el ejemplo de casa y calle, la distancia dice cuantas operaciones se han de realizar y de qué tipo (inserción, sustitución o borrado), así que lo que hace la clase con la mutación es obtener la distancia entre la cadena de referencia y la mutación. Después de obtener la distancia y el número de operaciones, se representa cada símbolo de cada operación cómo un símbolo diferente que será parte de la anotación.

Con lo dicho se puede definir tres símbolos (o tres *mappings*) por cada símbolo dependiendo de la operación que se haya realizado, que pueden ser inserción *i*, borrado *e* y sustitución *s*:

$$m_{m-i} = \{(A, t), (C, y), (G, u), (T, i)\}$$

$$m_{m-e} = \{(A, g), (C, h), (G, j), (T, k)\}$$

$$m_{m-s} = \{(A, a), (C, s), (G, d), (T, f)\}$$

Para aclarar lo dicho anteriormente se va a proceder a exponer un ejemplo. Supongamos que tenemos la cadena mutada *(AA, AC, CC)* y la cadena de referencia *TTGT*. Las operaciones entre *TTGT* y *AC* son dos eliminaciones y dos sustituciones, que serían las siguientes:

- Eliminar la letra *T*.
- Eliminar la letra *T*.
- Reemplazar la *G* por una *A*.

- Reemplazar la *T* por una *C*.

Así que la mutación anotada quedaría como:

$$S'_m = kkas$$

Por lo que, la cadena *(AA, AC, CC)* quedaría como:

$$S' = (qq, kkas, xx)$$

10.1.1.3. Escritura en archivo

La escritura en un archivo se podrá hacer de diferentes formas, pues se programará para que se pueda configurar en qué forma se mostrará el archivo, pero esto será explicado en el punto donde se explicarán las clases desarrolladas.

En el proyecto, el fichero contendrá pares de secuencias, una anotada y otra no para poder generar las muestras de aprendizaje y de test. Cada par ocupará dos líneas, una para la secuencia original y otra para la secuencia mutada.

De esta forma podrán obtenerse en la fase de entrenamiento. El archivo generado se tendrá como extensión *pvcf*, que significa vcf analizado (***p**arsed **vcf***).

10.1.2. Entrenamiento

La fase de entrenamiento se ha dividido en tres pasos, que son la obtención de datos de entrenamiento y test desde el archivo generado en la fase anterior, el entrenamiento del modelo y el proceso para guardar el modelo en un fichero, para que pueda ser utilizado en otro momento o por otro usuario.

Esto se ha hecho así para facilitar la tarea de realización de los *tests* y permitir más dinamismo en el momento de cambiar o escalar las funcionalidades. Para esta fase se han definido varias clases, una llamada *AbstractModel*, que es una representación de todos los métodos que debe tener definido un modelo para que pueda ser usado por la librería y una llamada *KTSSModel*, que es la clase que implementa el modelo de lenguajes k-explorables en sentido estricto.

La importancia de la clase *AbstractModel* será explicada en el punto donde se explican las clases, pues es la base para que la librería funcione correctamente en el apartado de generación de modelos.

En los siguientes puntos se va a proceder a explicar las acciones realizadas en cada paso.

10.1.2.1. Obtención de datos de entrenamiento y *test*

Para obtener los datos de *test* y de entrenamiento, *training* de ahora en adelante, se usará la clase *MutationParser*, que permitirá obtener y recuperar las secuencias que estén presentes en el archivo *pvcf* mediante una lista que será creada en la clase *KTSSModel*.

Para esto se leerá las líneas del archivo y se agruparán por pares. Estos pares contendrán una secuencia original y una secuencia anotada, ambas secuencias sin procesar, y serán almacenados en la clase *KTSSModel* en un formato de lista.

Más adelante será necesario procesar cada secuencia para adaptarla a la tarea en la que sea necesitada, ya sea para realizar *test* o *training*. La clase encargada de esto sería la clase que *MutationParser*, que tendría un método que será usado para procesar datos *de training* y otro para procesar datos de *test*. Esto último será común a todas las clases que hereden de *ParserVCF,* pues se pensó esto para que se pudiera adaptar a cualquier tipo de modelo. En el punto de descripción de las clases se explicará con más detalle.

Siguiendo con la preparación de datos, se procedería a obtener los datos de *training* que serán usados en el entrenamiento del modelo. Para ello se usaría un método definido en la clase *ParserVCF*, que permitiría obtener la información procesada para ser introducida como datos de entrenamiento.

10.1.2.2. Entrenamiento del modelo

A continuación se usarían estos datos en el método de *training* definido en *KTSSModel* para generar el modelo. Este método recibe como entrada los datos para entrenarse y una serie de parámetros (en el caso de lenguajes k-explorables el parámetro k) y devolverá el modelo generado en formato JSON.

El modelo generado será un AFD con una capa estocástica. En un principio iba a ser simplemente un AFD normal, pero en la fase de pruebas y experimentación se decidió retroceder y añadir la capa estocástica, para asignar una probabilidad a cada transición del autómata.

El método de *training* deberá ser implementado como una propiedad que herede del modelo *AbstractModel*, para que pueda ser usado por la librería y ajustado con parámetros enviados por la consola.

10.1.2.3. Guardado

Cuando se dispone del modelo entrenado se guardará en un fichero usando una función *saver* de la clase *KTSSModel*, que se encarga de introducir en fichero JSON todos los datos del modelo.

10.1.3. Validación

En esta última fase se pone a prueba el modelo con los datos de *test* generados y una clase llamada *KTSSValidation* o *KTSSViterbi*. Cada modelo se encarga de que a partir de una cadena sin anotar, es decir, sólo con los símbolos A, C, G y T genere una cadena anotada con los símbolos definidos en *MutationParser*.

De forma similar a la fase de entrenamiento, esta fase necesitará que las cadenas de test sean procesadas para que puedan ser usadas por el validador. Por eso se usará un método implementado en la clase *MutationParser*.

El proceso consistiría en obtener un par de muestras, donde cada par es una secuencia sin anotar y otra anotada, para que después de generar una secuencia anotada con el modelo, compararla con la secuencia anotada original y obtener un porcentaje de acierto.

Con todos los porcentajes de acierto se generaría un porcentaje total de acierto con el que se podría observar el rendimiento del modelo.

10.2. Clases principales

En este apartado se van a mostrar las clases principales que componen la base del proyecto, se va a exponer su uso y sus características así como la relación, si la tiene, con la clase *Runner*, que es la que se encarga junto con *ArgumentParser* de manejar la configuración que será enviada por consola.

Antes de empezar es necesario explicar las principales partes de la configuración que se puede enviar por consola, que son las que se modificarán en el método *test* del runner y las que se usarán en la generación y prueba del modelo:

- *-s*: Especifica el directorio donde se guardarán los archivos generados por la ejecución.
- *-p_p*: El tamaño del prefijo que tendrán las secuencias que se obtendrán del archivo FASTA.
- *-p_s*: El tamaño del sufijo que tendrán las secuencias que se obtendrán del archivo FASTA.
- *-r*: El porcentaje en decimales de muestras que se usarán para *training*, teniendo en cuenta que se usará 1 - *-r* para muestras de *test.*
- *-vcf*: Ubicación del fichero VCF.
- *-fasta*: Ubicación del fichero FASTA.
- *-test*: Indica si se quiere ejecutar el método *test* en lugar del método *run*.

Con estas opciones se puede configurar una ejecución completa de generación y validación del modelo.

10.2.1. Runner

Esta clase es la base que permite configurar los parámetros del modelo, así como la especificación de los ficheros VCF y FASTA y donde se guardarán los resultados.

La clase posee dos métodos principales:

- *run*: Se encarga de obtener los argumentos de consola mediante la clase *ArgumentParser*, de generar los datos de *training* y *test* y de ejecutar las fases de obtención de datos, de entrenamiento y de validación.

  La capacidad de realizar todas estas acciones de forma automática radica en las clases *ParserVCF* y *AbstractModel* (y en sus clases de configuración de parámetros *AbstractParserArguments* y *AbstractModelArguemtns*), ya que parte de la funcionalidad que implementan o obligan a implementar es usada por el método.

- *test*: Realiza la ejecución de la clase run pero con diferentes configuraciones y guardar en un archivo la configuración elegida y su resultado.

  Este método es usado para elegir la mejor configuración del modelo una vez acabada la implementación.

10.2.2. MutationParser

Esta clase se encarga de obtener y transformar datos de un fichero VCF y un fichero FASTA. Para esto se usa el método *generate_sequences*, que se encarga de generar el fichero *pvcf* con todas las mutaciones presentes en el fichero VCF. Para que este método funcione se debería implementar el método *method*, que es el que se encarga de anotar una secuencia. En el caso de la clase, en el método *method* se realizaría el proceso explicado en el punto 11.1.1.2.

A parte, esta clase implementa varios métodos que serán usados por el modelo para obtener tratar secuencias que vengan del archivo *pvcf*:

- *retrieve_sequence*: Obtiene la secuencia en formato tupla a partir de una secuencia en formato cadena.
- *retrieve_string_sequence*: Transforma una secuencia en formato cadena a partir de otra que venga de un archivo *pvcf*.

Estos métodos existen debido a que las secuencias son escritas en el fichero *pvcf* de una manera especial para que puedan ser tratadas después de diferentes formas. Por ejemplo, la secuencias anotadas se escriben en el archivo separando prefijo, sufijo e infijo con un espacio y separando los símbolos con el carácter '-':

secuencia_original\np-r-e-f-i-j-o i-n-f-i-j-o s-u-f-i-j-o

Y esa secuencia debe ser tratada de diferentes formas si se quiere transformar a formato tupla o a formato cadena juntando todos los caracteres.

10.2.3. FastaReader

Esta clase representa un archivo FASTA y permite acceder a su información. El principal método usado de esta clase es *sequence*, que a partir de un cromosoma, unos nucleótidos, su posición y los tamaños de prefijo y sufijo devuelve una secuencia con la información proporcionada del archivo FASTA.

Este método es el usado por la clase *MutationParser* para obtener las secuencias del archivo FASTA y generar las anotaciones.

10.2.4. KTSSModel

Esta clase es una de las más importantes, pues tiene la tarea de generar y guardar el modelo, así como relacionar el *Parser* utilizado (*MutationParser* en nuestro caso) y el validador con la clase *Runner*, pues esta clase llama a métodos del modelo para la ejecución. A continuación se va a proceder a explicar los métodos que son importantes para el funcionamiento de la clase *Runner*.

- *tester*: Método que sirve para realizar las validación del modelo.
- *trainer*: Método que sirve para realizar el entrenamiento del modelo.
- *saver*: Método que sirve para guardar el modelo en un fichero
- *model*: Atributo que contiene el modelo generado.
- *shuffle_samples*: Método que mezcla todas las muestras.

En la clase se especifica cómo será el modelo, tanto como serán las anotaciones especificando el *Parser* que se va a utilizar como el validador que mostrará el porcentaje de acierto del modelo, es el nexo de unión entre todas las clases principales.

10.2.5. KTSSValdiator - KTSSViterbi

La clase implementa las funcionalidades que permiten realizar las tareas de validación y generación del grado de acierto del modelo generado. Se han utilizado dos clases para anotar secuencias, una se basa en AFWK con una capa estocástica y otro se centra en un AFD con una capa estocástica y seleccionado las transiciones mediante el algoritmo de Viterbi.

También se encarga de generar anotaciones nuevas a partir de secuencias sin anotar usando el método *annotate_sequence*. Este método es usado también en el método *generate_distances*, que a partir de unas muestras de *test* genera el porcentaje de acierto del modelo.

En el caso de *KTSSViterbi*, se genera la cadena anotada por máxima verosimilitud usando el algoritmo de Viterbi, en el caso de *KTSSValidator* se anota eligiendo la transición con máxima probabilidad en cada fase del proceso de anotación anotación.

Para la gestión de los autómatas se crearon dos clases auxiliares, una *DFA* que representaba un AFD y un WatsonCrickAutomata, que permitía generar un AFWK a partir de un AFD y una serie de *mappings* y contenía un método para anotar una secuencia a partir de elegir la transición con más probabilidad en cada paso del análisis de la cadena.

10.3. Ejecución solución

A continuación se va a mostrar un ejecucion completa, tanto los valores que se introduzcan por consola como los *logs* que muestre el *logger* y el *porcentaje* de acierto del modelo:

```
python3 init.py -r 0.90 -steps 1 -o pm -vcf /opt/UPV/TFG/src/example/datosR1.vcf -fasta /opt/UPV/TFG/src/example/hg19.fa.gz -s /opt/UPV/TFG/src/example/  -p p 20 -p s 20 -p m -ao -m ktss
-k 3 -aoval -min -amval -sd
2021-06-29 12:15:05 INFO     Loading vcf file
2021-06-29 12:15:05 INFO     Loading fasta file
2021-06-29 12:15:05 INFO     Loading fasta information
2021-06-29 12:15:05 INFO     0it [00:00, ?it/s]
2021-06-29 12:15:05 INFO     136005it [00:00, 1359986.45it/s]
2021-06-29 12:15:05 INFO     297784it [00:00, 1511597.49it/s]
2021-06-29 12:15:05 INFO     464263it [00:00, 1581525.10it/s]
2021-06-29 12:15:06 INFO     630200it [00:00, 1612234.55it/s]
2021-06-29 12:15:06 INFO     797509it [00:00, 1634155.64it/s]
2021-06-29 12:15:06 INFO     960925it [00:00, 1632955.75it/s]
2021-06-29 12:15:06 INFO     1124221it [00:00, 1492181.21it/s]
2021-06-29 12:15:06 INFO     1287438it [00:00, 1534019.94it/s]
2021-06-29 12:15:06 INFO     1453151it [00:00, 1570849.70it/s]
```

Imagen 7: Parte 1 de la ejecución usando Viterbi

```
2021-06-29 12:15:43 INFO     60675307it [00:37, 1638155.64it/s]
2021-06-29 12:15:43 INFO     60839141it [00:37, 1623789.14it/s]
2021-06-29 12:15:43 INFO     61001557it [00:37, 1602334.88it/s]
2021-06-29 12:15:43 INFO     61164077it [00:37, 1609069.54it/s]
2021-06-29 12:15:43 INFO     61328855it [00:37, 1620538.42it/s]
2021-06-29 12:15:43 INFO     61490965it [00:37, 1607541.12it/s]
2021-06-29 12:15:43 INFO     61659266it [00:37, 1629915.94it/s]
2021-06-29 12:15:43 INFO     61825934it [00:38, 1640839.00it/s]
2021-06-29 12:15:43 INFO     61990862it [00:38, 1643336.90it/s]
2021-06-29 12:15:43 INFO     62155234it [00:38, 1643132.24it/s]
2021-06-29 12:15:44 INFO     62322349it [00:38, 1651491.93it/s]
2021-06-29 12:15:44 INFO     62487820it [00:38, 1652438.21it/s]
2021-06-29 12:15:44 INFO     62653079it [00:38, 1621914.70it/s]
2021-06-29 12:15:44 INFO     62743362it [00:38, 1622896.48it/s]
2021-06-29 12:15:44 INFO     Loading finalized

2021-06-29 12:15:44 INFO     Parsing sequences using mutation-type
2021-06-29 12:15:44 INFO     0it [00:00, ?it/s]
2021-06-29 12:15:44 INFO     358it [00:00, 3578.60it/s]
2021-06-29 12:15:44 INFO     417it [00:00, 3694.59it/s]
2021-06-29 12:15:44 INFO     Parsing finalized

2021-06-29 12:15:44 INFO     ########################################
2021-06-29 12:15:44 INFO     Validating step 0
2021-06-29 12:15:44 INFO     ########################################
2021-06-29 12:15:44 INFO     Training model
2021-06-29 12:15:44 INFO     Generating alphabet
2021-06-29 12:15:44 INFO     Dviding strings into greater or lower than 3
2021-06-29 12:15:44 INFO     Generating prefixes and suffixes
2021-06-29 12:15:44 INFO     0%|          | 0/375 [00:00<?, ?it/s]
2021-06-29 12:15:44 INFO     100%|##########| 375/375 [00:00<00:00, 7673.11it/s]
2021-06-29 12:15:44 INFO     Generating states from prefixes
2021-06-29 12:15:44 INFO     0%|          | 0/375 [00:00<?, ?it/s]
2021-06-29 12:15:44 INFO     100%|##########| 375/375 [00:00<00:00, 162335.02it/s]
2021-06-29 12:15:44 INFO     Generating states from infixes
2021-06-29 12:15:44 INFO     0%|          | 0/14683 [00:00<?, ?it/s]
2021-06-29 12:15:44 INFO     100%|##########| 14683/14683 [00:00<00:00, 380018.05it/s]
2021-06-29 12:15:44 INFO     Remove repeated and empty states
2021-06-29 12:15:44 INFO     Training finalized

2021-06-29 12:15:44 INFO     Generating validation data
2021-06-29 12:15:44 INFO     0%|          | 0/42 [00:00<?, ?it/s]
2021-06-29 12:15:44 INFO     83%|########3 | 35/42 [00:00<00:00, 347.05it/s]
2021-06-29 12:15:44 INFO     100%|##########| 42/42 [00:00<00:00, 347.41it/s]
2021-06-29 12:15:44 INFO     ########################################
2021-06-29 12:15:44 INFO     Model accuracy: 39.0075014426 %
2021-06-29 12:15:44 INFO     ########################################
```

Imagen 8: Parte 2 de la ejecución usando Viterbi

Como se puede ver, la ejecución va mostrando los avances usando mensajes generados por el *logger* y al final muestra el porcentaje de acierto llamado *accuracy*. La solución mostrada se basa en el algoritmo de Viterbi que, a partir de una secuencia sin anotar, elige el camino de máxima verosimilitud del AFD generado por la fase de entrenamiento.

La ejecución de la solución usando un AFWK elige la transición que tenga máxima probabilidad en cada paso de anotación de una cadena y da resultados muy parecidos, aunque la opción de usar AFWK y elegir la máxima probabilidad proporciona de media un 5% de acierto mayor.

10.4. Tests

Para la comprobación de que todos los métodos funcionan correctamente, se recurrió a la metodología TDD, con lo que en total se realizaron 139 tests unitarios para todas las clases y métodos que se implementaron.

La ejecución de los tests se realizó con pytest y el resultado de la ejecución es la siguiente:

```
poetry run pytest
============================================================================== test session starts ==============================================================================
platform linux -- Python 3.8.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /opt/UPV/TFG
collected 139 items

src/dataStructures/tests/test dfa.py .........                                                                                                                          [  6%]
src/dataStructures/tests/test dfaStochastic.py .....                                                                                                                    [ 10%]
src/dataStructures/tests/test wc automata.py ..........                                                                                                                 [ 17%]
src/fasta/tests/test chromosome.py ..............................                                                                                                       [ 41%]
src/fasta/tests/test fasta reader.py .....                                                                                                                              [ 45%]
src/model/tests/test ktss model.py ......................                                                                                                               [ 64%]
src/model/tests/test ktss validation.py ....................                                                                                                            [ 80%]
src/model/tests/test ktss viterbi.py ....                                                                                                                               [ 83%]
src/parser/tests/test extended parser vcf.py ............                                                                                                               [ 92%]
src/parser/tests/test mutation parser.py ....                                                                                                                           [ 95%]
src/utils/tests/test arithmetic.py ..                                                                                                                                   [ 97%]
src/utils/tests/test folders.py ...                                                                                                                                     [ 99%]
src/utils/tests/test genomics.py .                                                                                                                                      [100%]

============================================================================== 139 passed in 0.89s ==============================================================================
```

Imagen 9: La ejecución usando Viterbi

11. Experimentación

En esta fase se realizaron diferentes experimentaciones para elegir entre el modelo basado en *KTSSViterbi* o *KTSSValidation* los mejores parámetros. A continuación se van a mostrar los resultados de ambas ejecuciones en una gráfica, donde el eje X representa las claves tamaño_prefijo-tamaño_sufijo-k, donde k es el parámetro del modelo de lenguajes k-explorables:

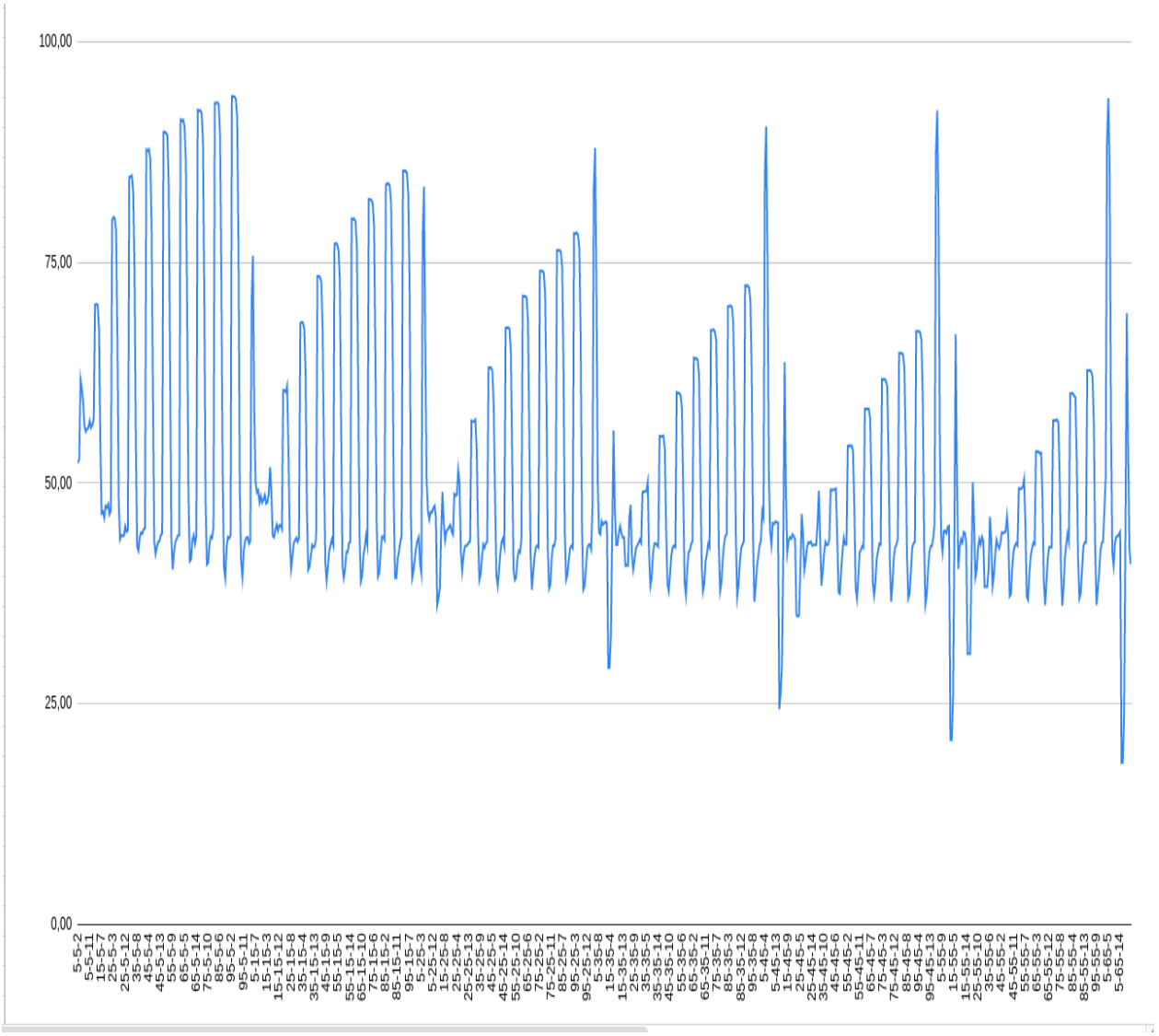

Imagen 10: Accuracy de cada parámetros con AWC

En la anterior figura se puede observar cómo el acierto del modelo del AFWK estocástico, que es el eje y, realiza oscilaciones entre puntuaciones altas y bajas. Las puntuaciones de acierto altas se presentan cuando el tamaño de prefijo es mucho mayor al tamaño de sufijo. Para mostrar esto, el acierto más alto obtenido en una serie de 800 pruebas tiene la forma 95-5-4.

Viendo esto en un archivo que se genera donde se muestran las distancias generadas entre la secuencia anotada original y la anotada por el modelo, se observa

que las mutaciones no se predicen correctamente, en cambio los prefijos si que los predice correctamente..

Respecto al modelo KTSSViterbi, los resultados son algo peores pero muestra el mismo patrón, los buenos resultados se muestran cuando el tamaño de prefijo es alto en comparación con el tamaño de sufijo. Al revisar las anotaciones, se puede observar que las mutaciones no se predicen correctamente. Por dar un ejemplo, la cota más alta se presenta en 90-15-2, con un 84.66%.

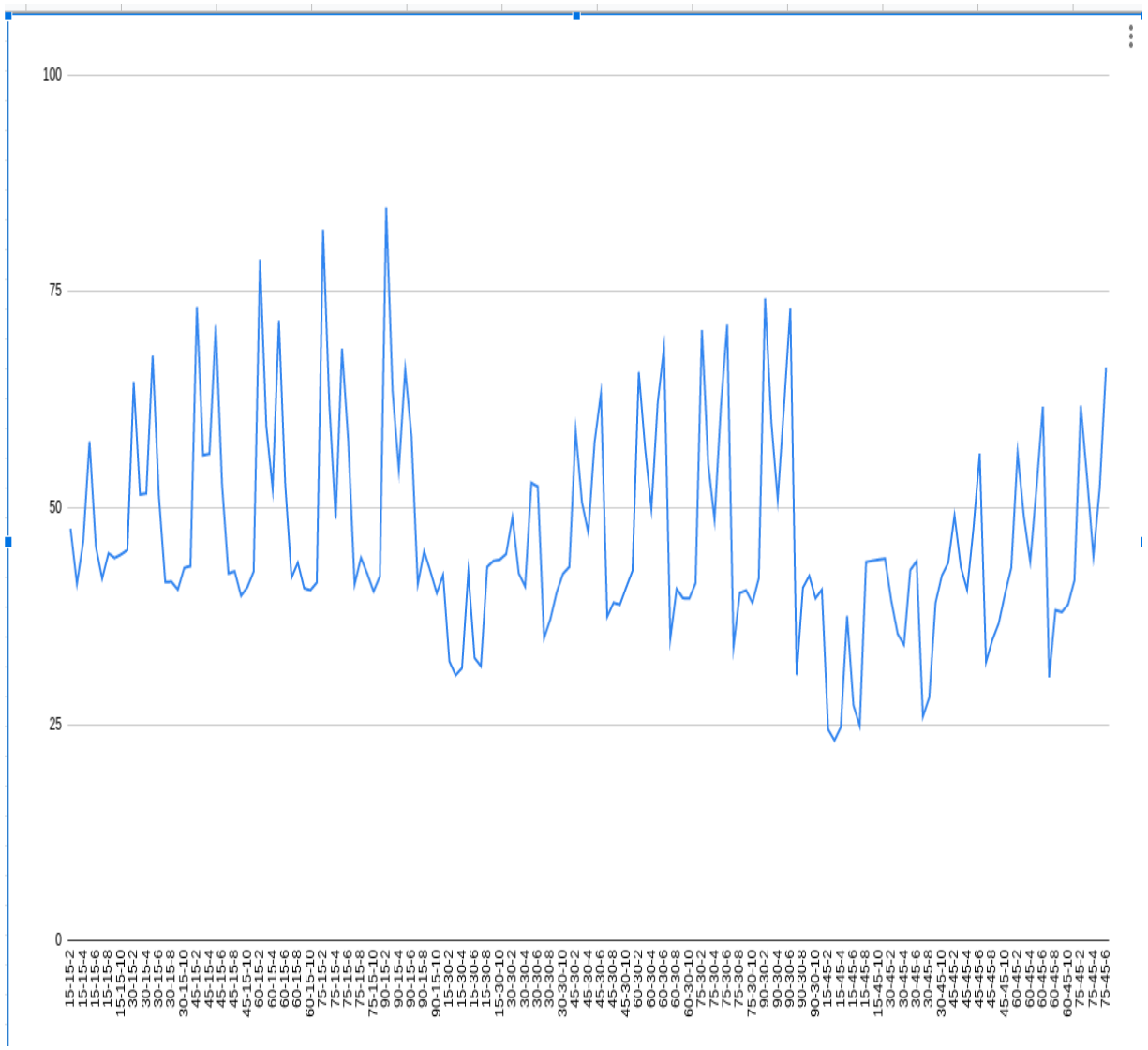

Imagen 11: Accuracy de cada configuración de parámetros con Viterbi

Gracias a estos resultados se pueden obtener varias conclusiones que serán explicadas en el siguiente punto, pero la más importante es que hay que cambiar el modelo o revisar los datos para poder asegurar si el problema es de falta de muestras (pues se ha probado con 400 muestras para *test* y *training*), de la forma en que se asignan las probabilidades a las transiciones o si simplemente el modelo de lenguajes k-explorables no es el adecuado para la tarea.

12. Conclusión

A la vista de los resultados, sería necesario contemplar otras vías para intentar mejorar el porcentaje de acierto en el caso de las predicciones de mutaciones, pues no se ha llegado al objetivo de más del 95%. Una de las posibles mejoras sería pensar un modo diferente a la hora de asignar las probabilidades a las transiciones, tanto en el caso del AFD como en el caso AFWK. Se podría asignar más peso a las transiciones en las que hubiera una mutación dependiendo de las transiciones que se han ejecutado hasta ese momento.

Otra posible vía sería cambiar el modo en que se anota la secuencia, pues puede llegar el momento en el que el modelo deba elegir entre varias posibles transiciones, para que pudiera elegir la mejor sin usar el modelo de Viterbi o la de máxima probabilidad, ya que estas formas son las que se usan.

Respecto a los demás objetivos, se han cumplido todos, pues el modelo queda generado en un fichero JSON y se permite anotar una cadena mediante un AFWK usando el método llamado *annotate_sequence*.

Siguiendo con las clases relacionadas con la extracción de secuencias de un fichero FASTA, se consiguió implementar una clase llamada *FastaReader* que permite obtener secuencias a partir del cromosoma y su posición. También se implementaron los *parsers*, que generaban cadenas anotadas con sus cadenas originales en ficheros con extensión *pvcf* a partir de archivos de mutaciones VCF y sus FASTA.

Finalmente, se creó la clase KTSSValidator que permite realizar una fase de pruebas al modelo y generar un porcentaje de acierto, así como un archivo Makefile que automatiza la forma en la que se ejecuta el *Runner*.

12.1. Problemas encontrados

Para la realización del proyecto se encontraron varios problemas que retrasaron la implementación o el análisis mientras se resolvían.

Uno apareció en la fase de análisis, y era la forma en la que se podían elegir los parámetros de la ejecución del programa. El resultado fue crear un clase llamada *ArgumentParser*, que permitía gestionar cómo se generaban los argumentos por consola especificando en las clases dónde se iban a utilizar y después añadiendolos al *Runner*. De esta forma, la librería *argparse* se encargaría de gestionarlos de forma correcta.

Otro problema que se descubrió en la fase de análisis es que no había una forma sencilla o ningún API sencilla que se pudiera utilizar para realizar consultas sobre secuencias de genomas humanos, así que se decidió realizar la clase FastaReader para gestionar esta problemática.

Finalmente, a la hora de generar y entrenar el modelo, cuando el número de muestras era muy grande y el parámetro K era mayor que diez, la memória RAM del ordenador se llenaba y no se permitía acabar el proceso, así que se decidió usar colas

en lugar de listas para la gestión de datos en la generación para aliviar la memória RAM. Al hacer esto se impactó de modo severo en el rendimiento, pues lo que costaba 5 minutos pasó a costar 25 minutos. Pero esto se podría arreglar paralelizando los bucles necesarios, ya que *Python* por defecto trabaja con un solo hilo del procesador.

12.2. Relación con el grado

Para el desarrollo del proyecto se han puesto en prácticas muchos de los conceptos vistos en las asignaturas del grado de ingeniería informática, especialmente los vistos en las siguientes asignaturas:

- TAL y LPPL - Teoría de autómatas y lenguajes formales y lenguajes de programación y procesadores del lenguaje: para entender las bases y los conceptos del algoritmo de los lenguajes k-explorables.

- PRG - Programación: Para dominar la forma en la que se gestionan la forma de programar correctamente y siguiendo buenas prácticas.

- EDA - Estructuras de datos y algoritmos**:** Para poder decidir que caminos escoger respecto a formas de organizar los datos ante problemas que pueden surgir, tanto de rendimiento como de espacio.

- ISW - Ingeniería del software: Para realizar de forma correcta las fases de análisis, gestión y *test*.

- APR y PER - Aprendizaje y percepción: Para entender cómo funcionan los paradigmas del aprendizaje automático y aplicar las nociones al proyecto.

- SAR - Sistemas de almacenamiento y recuperación: Para saber gestionar de forma correcta grandes cantidades de texto y cómo gestionar las relaciones entre cadenas.

Las anteriores son las principales asignaturas que han aportado el conocimiento para realizar el presente trabajo, pero todas las asignaturas del grado han aportado conocimiento y sabiduría para la correcta realización, tanto por los conceptos como por la variedad de posibilidades a elegir a la hora de plantear una solución a los problemas surgidos.

13. Trabajo futuro

El futuro de este proyecto podría radicar en mejorar el porcentaje de validación de datos del modelo de lenguajes k-explorables o demostrar que no es el modelo indicado.

Respecto a la extracción de datos de ficheros FASTA y VCF, se podrían paralelizar los bucles que obtienen los datos, permitiendo un desempeño más óptimo de las ejecuciones y mejorando el rendimiento. Esto último se podría aplicar también en la fase de entrenamiento, pues se podrían paralelizar varios bucles para hacer la generación más veloz.

Finalmente, se podría añadir una interfaz de usuario más amigable en formato web app y montar la librería en un servidor REST, así se podría generar un modelo y obtener datos desde ese servidor y devolverlos a la web app, para que cualquier tipo de usuario pudiera utilizar la librería y convertirla en una herramienta.

14. Referencias

de la Higuera, C. (2010). *Grammatical Inference Learning Automata and Grammars*. Cambridge. Cambridge University Press.

Deloitte. (2017). *Business impacts of machine learning*. https://www2.deloitte.com/content/dam/Deloitte/tr/Documents/process-and-operations/TG_Google%20Machine%20Learning%20report_Digital%20Final.pdf

P. García , E. Vidal and José Oncina.Learning Locally Testable Languages in the Strict Sense. In Proceedings of the First International Workshop on Algorithmic Learning Theory, ALT-90 (Tokyo, Japan). 1990. Edited by S. Arikawa, S. Goto, S. Ohsuga, T. Yokomory. pp 325-338. The Japanese Society for Artificial Intelligence. 1990.

Hopcroft, J., Ullman, J. (1979). *Theory, Languages, and Computation*. Addison Wesley Pub. Co.

Ignacio Herranz, J. (2011, 1 17). *TDD como metodología de diseño de software*. Retrieved 6 22, 2021, from https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/

Intel. (n.d.). *Cómo las empresas inteligentes avanzan con el aprendizaje automático*. Retrieved 6 27, 2021, from https://www.intel.la/content/www/xl/es/analytics/machine-learning/machine-learning-examples.html

Jamal, S., Khubaib, M., Gangwar, R., Grover, S., Grover, A., & Hasnain, S. E. (2020, 3 26). Artificial Intelligence and Machine learning based prediction of resistant and susceptible mutations in Mycobacterium tuberculosis. *Nature*, (5487).

Loria, S. (n.d.). *The Best of the Best Practices (BOBP) Guide for Python*. Retrieved 6 22, 2021, from https://gist.github.com/sloria/7001839

M. Bishop, C. (2006). *pattern recognition and machine learning* (2nd ed.). Springer.

Miranza. (n.d.). *Distrofia de retina*. Retrieved 6 27, 2021, from https://miranza.es/patologias/distrofia-de-retina/

Morales, A. (n.d.). *Lenguajes de programación para realizar ciencia de datos*. Retrieved 6 27, 2021, from https://mappinggis.com/2019/07/lenguajes-de-programacion-para-realizar-ciencia-de-datos/

Nagy, B., Parchami, S., & Mir-Mohammad-Sadeghi, H. (n.d.). *A New Sensing 5′ → 3′ Watson-Crick Automata Concept*. Turkey. Retrieved 6 27, 2021, from https://arxiv.org/pdf/1708.06469.pdf

Păun, G., Rozenberg, G., & Salomaa, A. (1998). *DNA Computing. Texts in Theoretical Computer Science*. Springer.

Sakakibara, Y. (2005). *Grammatical Inference in Bioinformatics* (Vol. 27). IEEE transactions on pattern analysis and machine intelligence. 10.1109/TPAMI.2005.140

Schwaber, K., & Sutherland, J. (2016). *La Guía Definitiva de Scrum: Las Reglas del Juego*. https://scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf#zoom=100

Sempere Luna, J. M. (2007). *On Local Testability in Watson-Crick Finite Automata*. In International Workshop on Automata for Cellular and Molecular Computing (ACMC07) (Budapest, Hungary). Gy. Vaszil (Ed.), pp 120-128. MTA SZTAKI. 2007.

Sempere Luna, J. M., & Heinz, J. (2016). *Topics in Grammatical Inference*. Springer. 10.1007/978-3-662-48395-4

topdoctors. (n.d.). *Enfermedades genéticas*. Retrieved 6 22, 2021, from https://www.topdoctors.es/diccionario-medico/enfermedades-geneticas

Vaino Aho, A., & David Ullman, J. (2007). *Principles of compiler design* (2nd ed.). Addison-Wesley.

Velogig. (2020, 4 1). *6 maneras en que Instagram utiliza Big Data, AI, y Machine Learning*. Retrieved 6 27, 2021, from https://velogig.com/6-maneras-en-que-instagram-utiliza-big-data-ai-y-machine-learning/

Wikipedia. (n.d.). *Ácido desoxirribonucleico*. Retrieved 6 22, 2021, from https://es.wikipedia.org/wiki/%C3%81cido_desoxirribonucleico

Wing-Kin, S. (2017). *Algorithms for next-generation sequencing*. CRC Press.