FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

Análisis de datos y aprendizaje automático en Haskell

Realizado por
Rafael Terán Parrado

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Octubre de 2024

Abstract

"*Data is the new oil*", this famous quote by *Clive Humby*, a renowned mathematician and pioneer in the analysis of large volumes of data to extract valuable consumer insights, said in 2006, may be more accurate today than it has ever been. Like oil, its true value comes within a proper refining and analysis.

In this project, we will provide a description on two of the most relevant methods in data analysis and machine learning: the *principal component analysis* and the *Naive Bayes classifier* are. We will then implement these methods in *Haskell* and demostrate their application through two examples.

Resumen

"*Los datos son el nuevo petróleo*", esta famosa cita de *Clive Humby*, un renombrado matemático y pionero en el análisis de grandes volúmenes de datos para extraer valiosos conocimientos sobre los consumidores, dicha en 2006, puede ser hoy más precisa que nunca. Al igual que en el petróleo, su verdadero valor se encuentra tras un adecuado refinamiento y análisis.

En este proyecto, proporcionaremos una descripción de dos de los métodos más relevantes en el análisis de datos y el aprendizaje automático: el *Análisis de Componentes Principales* y el *Clasificador Naive Bayes*. Posteriormente, implementaremos estos métodos en *Haskell* y demostraremos su aplicación a través de dos ejemplos.

Agradecimientos

A mi madre, a mi padre y a mi hermana.

Índice general

Índice de tablas

Índice de código

CAPÍTULO 1

Introducción

El manejo y la interpretación de grandes volúmenes de datos está a la orden del día, con el avance de la tecnología se hace posible recoger cantidades de datos nunca antes imaginables. Con este aumento han surgido nuevas técnicas y métodos de análisis, además de haberse perfeccionado los ya existentes, siempre buscando el objetivo de mejorar la visualización y facilitar la extracción de conclusiones. En el contexto del "*big data*", por mucho que se incremente la cantidad, si perdemos la "interpretabilidad" no habrá mejora alguna.

El término **aprendizaje automático** hace referencia a la rama de la inteligencia artificial basada en el desarrollo de modelos informáticos que en vez de limitarse a realizar tareas específicas concretas siempre de la misma manera, se nutren de bases de datos para ir afinando y mejorando en dar los resultados que más se parecen a la realidad. Se podría decir que los programas "aprenden". En este trabajo ahondaremos en dos técnicas concretas de diferente naturaleza, una de ellas formará parte del aprendizaje supervisado y partirá de un conjunto de ejemplos conocidos y previamente analizados, llamado *conjunto de entrenamiento*, para dar una respuesta, mientras que la otra será una técnica de aprendizaje no supervisado, donde no existirá dicho conjunto y el análisis será entorno a los datos por analizar, buscando relaciones entre ellos.

Hablaremos de la **clasificación de Naive Bayes** consistente en la asignación de una probabilidad a que un elemento a clasificar dado pertenezca a cada una de las clases posibles. El modelo necesitará una base de datos en las que existan ejemplos de elementos ya clasificados, por lo que se tratará de un método de aprendizaje supervisado.

Por otro lado, el **análisis de componentes principales** tiene un respaldo matemático más profundo, aunque la idea en la que se basa no es complicada de entender. En este caso no tendremos conjunto de entrenamiento, será un modelo de aprendizaje no supervisado, partiremos de una base de datos cuya naturaleza será de tener una gran cantidad de variables por observación. El método facilitará la obtención de posibles conclusiones, otorgando a las observaciones unas nuevas variables no correlacionadas que sustituirán a las originales perdiendo la mínima información en el proceso.

En este trabajo, previamente a cada modelo, daremos introducción a todos aquellos términos matemáticos, tanto estadísticos como algebraicos, que se verán inmiscuidos en su desarrollo teórico.

Continuaremos el trabajo con sus correspondientes implementaciones en librerías de

Haskell, un lenguaje funcional con amplia relación con el mundo matemático, terminando con unas aplicaciones más especificas que servirán de ejemplos de usos de las librerías implementadas.

CAPÍTULO 2

Clasificador Naive Bayes

El *clasificador Naive Bayes* es el primero de los algoritmos de *machine learning* que veremos. Fundamentado en el teorema de Bayes, es uno de los algoritmos de referencia en el campo de la clasificación automática, utilizando un método probabilístico sorprendentemente simple como fundamento teórico. Es conocido por ser una herramienta de clasificación usada principalmente en textos, sus utilidades suelen estar relacionadas con la detección de *spam* en correos electrónicos, o como en la aplicación práctica que veremos en el capítulo "**Aplicaciones**", con la clasificación de textos por idioma. Además de ser un muy buen punto de entrada al mundo del *aprendizaje automático* o *machine learning*.

2.1– Conocimientos probabilísticos previos

El objetivo de esta sección es introducir de manera leve varios conceptos probabilísticos que serán esenciales para el correcto entendimiento de la teoría detrás del Clasificador Naive Bayes. El objetivo de este ensayo no es explicar estos conceptos a fondo, y por lo tanto se recomienda a aquellos lectores menos familiarizados con el mundo de la estadística o aquellos que quieran informarse, que busquen información adicional sobre el tema. Esta pequeña introducción está basada en los apuntes de Francisco Montes Suay titulados "Introducción a la Probabilidad" (2007, Universitat de València), los cuales se encuentran referenciados en la bibliografía.

2.1.1. Experimento, resultado, espacio muestral y suceso

Introducimos algunos de los conceptos principales dentro de la teoría de la probabilidad.

Definición 2.1. Un *experimento aleatorio* es un proceso o fenómeno que, bajo condiciones controladas, puede reproducirse indefinidamente, pero dando un resultado específico que no se puede predecir con certeza antes de su realización.

Ejemplo 2.2. Un *experimento aleatorio* podría consistir en lanzar un dado perfecto o en sacar, sin mirar, una bola de una caja que contiene tres bolas cuya única diferencia es el color.

Definición 2.3. Un *resultado*, ω, de un experimento aleatorio es el valor o conjunto de valores obtenidos al realizar el experimento.

Ejemplo 2.4. Los *resultados* de los ejemplos anteriores serían un número del 1 al 6 en el caso del dado, o uno de los tres colores en el caso de las bolas.

Definición 2.5. Un *espacio muestral*, Ω, de un experimento aleatorio es el conjunto de posibles resultados que puede tener el experimento.

Ejemplo 2.6. Los *espacios muestrales* de los ejemplos anteriores serían $\Omega = \{1, 2, 3, 4, 5, 6\}$ para el dado y $\Omega = \{rojo, azul, amarillo\}$ para las bolas.

Definición 2.7. Un *suceso aleatorio*, A,B,..., es un subconjunto de un espacio muestral, normalmente caracterizado por una cualidad común. Cuando el resultado del experimento forma parte de un suceso aleatorio A, se dice que ha ocurrido o se ha realizado A.

Ejemplo 2.8. Unos *sucesos aleatorios* del ejemplo del dado podrían ser:

$$A = \{\text{Ha salido par}\} = \{\omega \text{ es par}\} = \{2, 4, 6\} \tag{2.1}$$

$$B = \{\text{Ha salido menor que 4}\} = \{\omega < 4\} = \{1, 2, 3\} \tag{2.2}$$

2.1.2. σ-álgebra de conjuntos

Para definir la probabilidad de un suceso aleatorio, tenemos que determinar algunas propiedades que deben poseer estos conjuntos. Por ejemplo, es lógico pensar que todo suceso y su complementario deben sumar una probabilidad de 1, expresándola en tanto por uno. O que la probabilidad de unir dos sucesos deberá ser la probabilidad de ambos sumadas.

A partir de ahora, para asegurarnos de que trabajamos sobre conjuntos de sucesos que sean coherentes con estas propiedades, cuando se hable de un suceso aleatorio, se estará hablando formalmente de un elemento de una σ-álgebra de conjuntos, una σ-álgebra de sucesos más concretamente.

Definición 2.9. Una *σ-álgebra de conjuntos* se define como una familia de conjuntos $\mathcal{A}$ sobre Ω tal que:

1. $\Omega \in \mathcal{A}$.
2. $A \in \mathcal{A} \Rightarrow A^c \in \mathcal{A}$.
3. $\{A_n\}_{n\geq 1} \subset \mathcal{A} \Rightarrow \bigcup_{n\geq 1} A_n \in \mathcal{A}$.

En la definición de suceso que hemos dado, es evidente que tanto el complementario de un suceso como la unión y la intersección de sucesos serían sucesos. Por lo que existe una compatibilidad entre ambas definiciones. El concepto de σ-álgebra será especialmente importante para asegurar la estabilidad de esas propiedades en sucesos no numerables.

Podemos observar que el conjunto de todos los sucesos, es decir, la familia de las partes de Ω, $\mathcal{P}(\Omega)$, tiene estructura de σ-álgebra.

2.1.3. Probabilidad

La idea de la probabilidad no es más que dar un número que asocie un suceso de un experimento aleatorio a la posibilidad de que se realice. Para ello se define una función sobre la σ-álgebra de sucesos, a esta función se le llama medida de probabilidad o probabilidad directamente.

Definición 2.10. Una *probabilidad*, P, definida sobre la una σ-álgebra $\mathcal{A}$ es una función tal que:

1. $P(A) \geq 0$ para todo $A \in \mathcal{A}$.
2. $P(\Omega) = 1$.
3. P es numerablemente aditiva, es decir, si $\{A_n\}_{n\geq 1}$ es una sucesión de sucesos de $\mathcal{A}$ disjuntos dos a dos, entonces

$$P\left(\bigcup_{n\geq 1} A_n\right) = \sum_{n\geq 1} P(A_n).$$

A la terna $(\Omega, \mathcal{A}, P)$ la denominaremos *espacio de probabilidad*.

Ejemplo 2.11. Volviendo al ejemplo del dado, la *función probabilidad* sería:

$$P(A) = P(\{\text{Ha salido el 5}\}) = P(\{\omega = 5\}) = P(\{5\}) = \frac{1}{6} = 0,166 \tag{2.3}$$

$$P(B) = P(\{\text{Ha salido par}\}) = P(\{\omega \text{ es par}\}) = P(\{2,4,6\}) = \frac{3}{6} = 0,5 \tag{2.4}$$

$$P(\Omega) = P(\{\text{Ha salido un número del 1 al 6}\}) = P(\{1,2,3,4,5,6\}) = \frac{6}{6} = 1 \tag{2.5}$$

2.1.4. Probabilidad condicionada

A la hora de calcular una probabilidad, a veces hay que tener en cuenta otros sucesos que ya ocurrieron, o considerar la posibilidad de que ocurran. Al existir la posibilidad de que esto cambie la probabilidad de un suceso, se debe definir la probabilidad condicionada.

Definición 2.12. Sea $(\Omega, \mathcal{A}, P)$ un espacio de probabilidad y sean A y B dos sucesos, cumpliendo $P(B) \neq 0$, definimos la *probabilidad* de A *condicionada* a B como:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \tag{2.6}$$

Ejemplo 2.13. Para un experimento consistente de sacar varias bolas al azar, sin devolverlas, de una caja donde hay dos bolas rojas y otras tres azules, siendo $R = \{\text{Sacar una bola roja}\}$ y $A = \{\text{Sacar una bola azul}\}$, sus *probabilidades* serían:

$$P(R) = \frac{2}{5} = 0,4 \qquad P(A) = \frac{3}{5} = 0,6 \tag{2.7}$$

Por lo que las *probabilidades condicionadas* tomarían la siguiente forma:

$$P(R \mid A) = \frac{2}{4} = 0,5 \qquad P(A \mid A) = \frac{2}{4} = 0,5 \tag{2.8}$$

$$P(R \mid R) = \frac{1}{4} = 0,25 \qquad P(A \mid R) = \frac{3}{4} = 0,75 \tag{2.9}$$

Tomando la definición de probabilidad condicionada, se puede deducir una expresión de la probabilidad de la intersección de dos sucesos en función de la probabilidad condicionada:

$$P(A \cap B) = P(A \mid B) \cdot P(B) \tag{2.10}$$

2.1.5. Independencia

En la cara opuesta de la idea que nos hizo pensar en la probabilidad condicionada, está la independencia de sucesos. Muchas veces dos o más sucesos no guardan ninguna relación entre sí, y que ocurra o deje de ocurrir uno o varios de ellos no afecta en la probabilidad del resto.

Definición 2.14. Sean A y B dos sucesos, decimos que A y B son *independientes* si y solo si:

$$P(A \cap B) = P(A) \cdot P(B) \tag{2.11}$$

Se puede observar que en el caso de que dos sucesos A y B sean independientes, sus probabilidades condicionadas coinciden con sus probabilidades simples:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) \cdot P(B)}{P(B)} = P(A) \tag{2.12}$$

También será importante extender el concepto a no solo una relación dual sino que un grupo de sucesos finito sean independientes entre sí. Llamaremos a esto independencia mutua.

Definición 2.15. Sea $\{A_1, ..., A_n\}$ una familia de sucesos, decimos que son *mutuamente independientes* si y solo si

$$P(\bigcap_{i=1}^{m} A_{k_i}) = \prod_{i=1}^{m} P(A_{k_i}) \tag{2.13}$$

siendo $\{k_1, ..., k_m\} \subseteq \{1, ..., n\}$

2.2– Teorema de Bayes

Ya con todos esos conceptos planteados y las notaciones claras, damos el paso a introducir el Teorema de Bayes, de donde saldrá toda la base teórica para sacar adelante el clasificador Naive Bayes.

El teorema de Bayes es uno de los conceptos principales en la estadística. Planteado por el matemático y reverendo inglés Thomas Bayes (1702-1761), se trata de la base de toda la estadística bayesiana, así como un bastión fundamental en todo el desarrollo de las técnicas de análisis de datos.

La idea en la que se fundamenta el teorema es bastante intuitiva y sigue un razonamiento que podemos tener en el día a día cuando se intenta hacer una predicción o tomar una decisión sobre un tema sin tener todos los datos.

Probabilidad a priori

El primer concepto que tomaremos es la **probabilidad a priori**, $P(A)$. Previo a todo juicio, todos tenemos una idea preconcebida o unas creencias iniciales previas a cualquier evidencia o información nueva. Normalmente está basada en experiencias pasadas o conocimiento general.

Por ejemplo, si nuestro suceso a estudiar (A) es si va a llover hoy. Si llovió ayer y estamos en otoño, hay una probabilidad a priori ($P(A)$) alta de que hoy llueva. Esta es una idea inicial previa a evidencias como lo son mirar por la ventana o mirar el pronóstico del tiempo.

Verosimilitud

Habiendo hecho una primera aproximación al problema, se pasa al concepto de **verosimilitud**, $P(B \mid A)$, en él, habiendo recogido alguna evidencia (B), se cuestiona la probabilidad de esta, tomando como cierto el primer acercamiento al problema (A). La verosimilitud es la probabilidad de haber observado la evidencia tomada si el primer juicio fue el acertado, es decir, dice cómo de coherente era la creencia inicial.

Volviendo al ejemplo donde se piensa que va a llover, si después de hacer la predicción se decide abrir la ventana y no se ve ninguna nube en el cielo, la verosimilitud diría: *"Si es cierto que hoy va a llover, ¿cómo de probable es que no esté nublado?"*. Estamos dando como cierto que hoy llueve (A) y vemos la probabilidad de que esté nublado sabiendo esto ($P(B \mid A)$).

Evidencia

La probabilidad de que lo observado sea cierto será la **evidencia o verosimilitud marginal**, $P(B)$. La importancia de analizar lo común que sea la nueva información recogida reside en ayudar a contextualizar la información y restarle o sumarle peso. La relación entre este dato y el resto será de proporcionalidad inversa, cuánto más probable sea la evidencia, menos importará en nuestra conclusión final.

En el ejemplo de la lluvia, si la observación fue ver nubes en el cielo (B), la verosimilitud marginal ($P(B)$) dependerá de cómo de común sea ver nubes en el cielo. Si se vive en Andalucía tendrá más importancia pues no siempre hay nubes y pueden significar que lloverá, sin embargo si se vive en Galicia ver nubes en el cielo es mucho más común y no es tan significativo porque siempre hay nubes, llueva o no.

Probabilidad a posteriori

La última, y fundamental variable en el teorema de Bayes, es la **probabilidad a posteriori**, $P(A \mid B)$. Es el resultado a estudiar y la creencia o conclusión final del problema. Sin más preámbulos, es la probabilidad de que se dé el suceso a estudiar dada una cierta información observada.

En nuestro ejemplo, la probabilidad a posteriori ($P(A \mid B)$) es la respuesta a cómo de probable es que la hipótesis tomada (A) sea cierta, en este caso, que hoy llovía, sabiendo que está nublado (B).

La misión principal del teorema de Bayes es ajustar la probabilidad a posteriori según la información recibida o preconcebida. Básicamente el teorema dice que la probabilidad a posteriori es el resultado de combinar la verosimilitud y la probabilidad a priori, siendo esto normalizado por la verosimilitud marginal o evidencia, para que el resultado sea una probabilidad.

De manera formal, esta es la formulación del **teorema de Bayes**:

Teorema 2.1. *Sean A y B dos sucesos, tal que $P(B) \neq 0$, se tiene:*

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)} \tag{2.14}$$

La demostración es tan sencilla como aplicar la definición de probabilidad condicionada:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B \mid A) \cdot P(A)}{P(B)} \tag{2.15}$$

2.3– Clasificación Naive Bayes

Contando con todos los conceptos matemáticos previos, nos adentraremos en el funcionamiento del Clasificador Naive Bayes. Podemos definir un clasificador como un método supervisado que permite entrenar un modelo para que determine la clase más probable que debemos asignar a un registro, en este caso, el método que seguirá estará basado en el Teorema de Bayes. El entrenamiento que se dará al modelo será un conjunto de una gran cantidad de ejemplos ya clasificados en sus respectivas clases, al que llamaremos **conjunto de entrenamiento**. Gracias a estos, podremos buscar patrones o similitudes entre los registros a clasificar y los elementos del conjunto de entrenamiento. Si un registro guarda muchas similitudes con elementos de entrenamiento clasificados en una clase concreta, este tendrá muchas probabilidades de pertenecer a esta misma clase.

Partiendo del teorema de Bayes, tendremos:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)} \tag{2.16}$$

Donde:

- A es el suceso de que un registro pertenezca a una clase concreta.
- B es el suceso de observar una característica específica en el registro.

Extendemos la fórmula a una versión donde tengamos una cantidad finita de sucesos dados, $B_1, B_2, ..., B_n$:

$$P(A \mid B_1, B_2, ..., B_n) = \frac{P(B_1, B_2, ..., B_n \mid A) \cdot P(A)}{P(B_1, B_2, ..., B_n)} \tag{2.17}$$

Suposición Naive

El nombre de la clasificación, *"Naive Bayes"*, proviene de la fusión entre el teorema de Bayes y el término *Naive*, palabra inglesa que significa *ingenuo*. Se le llama ingenuo al modelo por asumir que los datos que condicionan la probabilidad no tienen dependencia entre ellos, es decir, son mutuamente independientes.

Esto nos deja con la siguiente ecuación:

$$P(A \mid B_1, B_2, ..., B_n) = \frac{P(B_1 \mid A) \cdot P(B_2 \mid A) \cdot ... \cdot P(B_2 \mid A) \cdot P(A)}{P(B_1) \cdot P(B_2) \cdot ... \cdot P(B_n)} \tag{2.18}$$

Para nuestros intereses, no nos preocupa el valor real de la probabilidad que tenga un elemento de pertenecer a una clase u otra, simplemente queremos tener un valor orientativo que nos permita comparar probabilidades y quedarnos con la opción más probable, la probabilidad en sí no es significativa para el clasificador. Por lo tanto, observando que en nuestra ecuación el denominador no nos da una diferencia comparativa, ya que no depende de la clase, A, podemos simplificar la fórmula eliminando el denominador de la ecuación:

$$P(A \mid B_1, B_2, \ldots, B_n) \propto P(A) \cdot \prod_{i=1}^{n} P(B_i \mid A) \tag{2.19}$$

La clase más probable y que asignaremos a cada registro será aquella que maximice la ecuación anterior. Formalmente, si fuera $\mathcal{A} = A_1, A_2, ..., A_m$ el conjunto de clases a las que puede pertenecer un registro, la clase que nuestro modelo seleccionaría sería la siguiente:

$$A^* = \arg\max_{A \in \mathcal{A}} P(A) \cdot \prod_{i=1}^{n} P(B_i \mid A) \tag{2.20}$$

2.3.1. Suavizado de Laplace

En la teoría el resultado anteriormente presentado funciona perfectamente, pero en la práctica hay que tener en mente algunas consideraciones. Hay que ser consciente de que la base de datos que sirva como conjunto de entrenamiento no es infinita, está limitada, en muchos casos no contiene todos los posibles fenómenos que se relacionan con la clase a la que pertenecen. Al estar multiplicándose todas las probabilidades, con que una de las características del registro no se encuentre en el conjunto de entrenamiento, este tendrá probabilidad nula y hará que el producto sea nulo, imposibilitando la correcta clasificación.

Esto puede entenderse mejor con un ejemplo. Uno de los posibles usos que puede tener un clasificador Naive Bayes es el de filtrar reseñas o comentarios de un establecimiento en función de si son positivos, negativos o neutros. Para ello en nuestra base de datos tendremos palabras positivas como pueden ser "rico", "barato", "estupendo", etc., provenientes de mensajes positivos que recogimos en la base de datos, pero no podemos tener todas las palabras existentes en castellano. Si intentamos clasificar un comentario que dice "Fui con mi prima y estuvo todo estupendo", existe la posibilidad de que la palabra "prima" no se encuentre en el conjunto de palabras que se relacionan con los mensajes positivos, es decir, que en ninguna reseña positiva del conjunto de entrenamiento se mencione la palabra "prima".

$$P(\text{"prima"} \mid \text{Reseña positiva}) = 0 \tag{2.21}$$

El comentario es evidentemente positivo pero por una sola palabra se arruina todo el proceso. A este problema se le llama **Problema de Cero Frecuencia**.

Una manera de resolver este error es mediante una técnica llamada **suavizado de Laplace (Laplace Smoothing)**. El suavizado de Laplace se suele usar en estadística y aprendizaje automático para solucionar problemas de estimación de probabilidad como en el caso en el que nos encontramos.

La solución que ofrece el suavizado de Laplace es añadir una constante en el cálculo de cada probabilidad, haciendo que no existan probabilidades nulas pero sin perder la proporción. Partiendo de que el cálculo de la probabilidad sin suavizado es:

$$P(B_i \mid A) = \frac{nAB_i}{nA} \tag{2.22}$$

Donde:

- nAB_i es el número de veces que se repite el suceso B_i en el conjunto de entrenamiento y que está asociado a la clase A.
- nA es el número de sucesos sin repetición que hay en el conjunto de entrenamiento asociados a la clase A.

La modificación a la probabilidad para el suavizado sería la siguiente:

$$P'(B_i \mid A) = \frac{nAB_i + \alpha}{nA + \alpha \cdot N} \tag{2.23}$$

Donde:

- $\alpha > 0$ es una constante (normalmente se coge $\alpha = 1$ por simplicidad).
- N es el número total de sucesos sin repetición que hay en el conjunto de entrenamiento.

Llamaremos **probabilidad suavizada** a P', tras implementarla en el nuevo y definitivo modelo del clasificador, nos quedaría la siguiente ecuación:

$$A^* = \arg\max_{A \in \mathcal{A}} P(A) \cdot \prod_{i=1}^{n} P'(B_i \mid A) \tag{2.24}$$

CAPÍTULO 3

Análisis de componentes principales

La segunda técnica de análisis de datos que abordaremos es el *análisis de componentes principales* (PCA). Para enfrentar el desafío de analizar desbordantes cantidades de datos con abundantes variables diferentes que hacen la comprensión de los datos en crudo prácticamente imposible, una herramienta ampliamente utilizada y probada es el *análisis de componentes principales*, consistente en reducir la "dimensionalidad" del conjunto de datos, tratando de minimizar la perdida de información en el proceso. El objetivo será determinar un nuevo número, mucho menor, de variables nuevas que condensen la información que teníamos en las antiguas variables, de las cuales muchas es normal que sean redundantes o poco importantes. Esto se llevará a cabo utilizando instrumentos del álgebra lineal, como lo es la descomposición en valores y vectores propios de una matriz.

3.1– Introducción estadística

Empezaremos esta incursión en el *análisis de componentes principales* partiendo de las bases teóricas que lo sustentan. En esta primera sección daremos las claves para la comprensión de conceptos principales en estadística como lo son la **varianza** y la **covarianza**, para terminar con una ampliación de este último, la **matriz de covarianzas**. Toda la introducción se encuentra respaldada por el libro "An Introduction to Probability Theory and Its Application" de William Feller, debidamente citado en la bibliografía, se recomienda su lectura para aquellos que quieran ampliar sus conocimientos en este campo, concretamente trataremos el noveno capítulo "Random Variables; Expectation".

Variables aleatorias

Durante la presentación del *clasificador Naive Bayes*, se definió el concepto de **experimento aleatorio**, así como el de resultado de un suceso aleatorio, recordemos que se trata del valor o conjunto de valores obtenidos al realizar un experimento aleatorio. Volviendo a esas ideas definimos un primer resultado íntimamente relacionado con ellas:

Definición 3.1. Una *variable aleatoria*, X, Y,..., es una función que devuelve o asigna un valor al resultado de un experimento aleatorio.

Ejemplo 3.2. Ejemplos de *variables aleatorias* pueden ser el número de personas que entran cada hora en una tienda (X = número de personas), o un 0 o un 1 si se saca cara o cruz tirando una moneda ($Y = 1$ ó 0).

Función de probabilidad

Rescatamos un concepto del capítulo anterior, la **probabilidad**, P, para dar el siguiente resultado:

Definición 3.3. Sea X una variable aleatoria y $x_1, x_2, \ldots$ todos los posibles valores que asigna, definimos la *función de probabilidad* (o *función de masa de probabilidad*) de una variable aleatoria X, como:

$$f(x_i) = f_X(x_i) = P[X = x_i] \tag{3.1}$$

La probabilidad de $X = x_i$, $P[X = x_i]$, estará bien definida al haber definido la probabilidad sobre σ-álgebras de sucesos y siendo $X = x_i$ el suceso de que se dé un resultado que se relacione con x_i. La idea de la función es simple, relaciona cada valor de la variable aleatoria con la probabilidad de que este se dé.

3.1.1. Esperanza, varianza y covarianza

Una vez definidas las variables aleatorias, describiremos algunas de las herramientas para su análisis.

Esperanza

Yendo más allá en el análisis de las variables aleatorias, un problema razonable sería preguntarse, ¿qué debemos esperar de cada variable aleatoria?, es decir, cuál sería una aproximación del valor que dará la variable aleatoria, o cual es el **valor esperado** de ella.

Para dar respuestas a esta preguntas se define la *esperanza* de una variable aleatoria que será similar a la noción popular de la *media* de un conjunto de datos, nos dará una idea de los valores que puede tomar la variable aleatoria.

Definición 3.4. Sea X una variable aleatoria, $x_1, x_2, \ldots$ todos los posibles valores que asigna y f su función de probabilidad, definimos la *esperanza* de una variable aleatoria X, como:

$$E[X] = \sum_{i \geq 1} x_i \cdot f(x_i) \tag{3.2}$$

Siempre en caso de que la suma converja absolutamente, en caso contrario, diremos que X no tiene esperanza finita.

Varianza

Además de la esperanza, otra herramienta estadística básica que debemos conocer es la variabilidad de la variable aleatoria. Esta nos dará una visión más profunda de los valores de la variable aleatoria. Muchas veces dos variables aleatorias pueden tener una esperanza similar pero en realidad, al tratarse de una media, los valores de una de ellas pueden ser muy estables en la esperanza y los de la otra tener grandes oscilaciones. Al valor que nos dará una imagen de esta estabilidad o variabilidad, lo definiremos como:

Definición 3.5. Sea X una variable aleatoria con esperanza finita, definimos la *varianza* de X, como:

$$Var(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2 \tag{3.3}$$

Expresado de otra manera, la varianza es la esperanza de la desviación al cuadrado de la esperanza de la variable aleatoria.

Al elevarse al cuadrado, la varianza siempre tendrá valor positivo y será mayor cuanto menos estable sea la variable y más disparejos sean los valores.

Covarianza

Siguiendo la misma línea de razonamiento, podemos llegar a la idea de **covarianza**. Será de vital importancia, cuando se trabaje con más de una variable, poder estimar el grado de similitud o disimilitud que existe entre dos variables, más concretamente sobre sus valores. La siguiente definición será capaz de cuantificar esta propiedad:

Definición 3.6. Sean X e Y variables aleatorias con esperanza finita, definimos la *covarianza* de X e Y, como:

$$Cov(X, Y) = E[(X - E[X]) \cdot (Y - E[Y])] \tag{3.4}$$

Una manera sencilla de entender la relación entre la varianza y la covarianza es pensar que la varianza es la covarianza de una variable aleatoria consigo misma, $Cov(X, X) = Var(X)$.

Los valores de la covarianza oscilarán en torno al cero, siendo $Cov(X, Y) = 0$ el caso en el X e Y son *incorreladas*, es decir no tienen similitud o disimilitud. Un mayor valor positivo indica mayor grado de semejanza, mientras que un menor valor negativo todo lo contrario.

3.1.2. Vector aleatorio

En muchas ocasiones tendremos que un mismo experimento nos da muchos resultados, lo que nos dará muchas variables aleatorias distintas, para operar con ellas se organizarán dentro de una estructura con forma de vector que llamaremos *vector aleatorio*.

Definición 3.7. Sean $X_1, X_2, \ldots, X_n$ n variables aleatorias, definiremos un *vector aleatorio* como una colección de n variables aleatorias, expresado como:

$$\underline{X} = (X_1, X_2, \ldots, X_n)^T \tag{3.5}$$

Esperanza de un vector aleatorio

Ampliaremos la noción de esperanza de una variable aleatoria a la de un vector aleatorio sin demasiado problema:

Definición 3.8. Sean $\underline{X} = (X_1, X_2, \ldots, X_n)^T$ un vector aleatorio de n variables aleatorias con esperanza finita, definiremos la *esperanza* de un vector aleatorio como el vector de las esperanzas de sus componentes:

$$E[\underline{X}] = (E[X_1], E[X_2], \ldots, E[X_n])^T \tag{3.6}$$

Matriz de covarianzas

Más interesante que la generalización a varias dimensiones de la esperanza, lo es la de la varianza y la covarianza. En este caso no tendremos un resultado con forma de vector, sino que optaremos por una matriz.

Definición 3.9. Sean $\underline{X} = (X_1, X_2, \ldots, X_n)^T$ un vector aleatorio de n variables aleatorias con esperanza finita, definiremos la *matriz de covarianzas*, Σ, de un vector aleatorio como la matriz:

$$\Sigma = Var(\underline{X}) = E[(\underline{X} - E[\underline{X}]) \cdot (\underline{X} - E[\underline{X}])^T] \tag{3.7}$$

O expresado de manera más explícita:

$$\Sigma = Var(\underline{X}) = \begin{pmatrix} Var(X_1) & Cov(X_1, X_2) & \ldots & Cov(X_1, X_n) \\ Cov(X_2, X_1) & Var(X_2) & \ldots & Cov(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(X_n, X_1) & Cov(X_n, X_2) & \ldots & Var(X_n) \end{pmatrix} \tag{3.8}$$

donde Var y Cov son respectivamente la varianza y la covarianza de las variables aleatorias del vector $\underline{X}$ y sabiendo que $Cov(X_i, X_i) = Var(X_i)$.

Propiedades de la matriz de covarianza

Para complementar la definición, mencionaremos las siguientes propiedades de la matriz de covarianza:

- Σ es una matriz simétrica
- Σ es semidefinida positiva
- $Var(\underline{X} \cdot a) = a^T \cdot Var(\underline{X}) \cdot a$, con $a \in \mathbb{R}^n$

3.1.3. Inferencia muestral

Para terminar la introducción estadística, utilizamos esta última subsección para tratar ciertos aspectos dentro del territorio de la inferencia estadística. Veremos cómo extrapolar los conceptos de varianza o matriz de covarianza cuando partimos de una toma de muestras aleatorias, empezando por definir a qué nos referimos exactamente con esto.

Muestra aleatoria

Definición 3.10. Sea $x_1, x_2, \ldots, x_n$ observaciones o resultados concretos de variables aleatorias idénticas definidas en un mismo espacio de probabilidad. Diremos que $x_1, x_2, \ldots, x_n$ constituyen una *muestra aleatoria* de dimensión n si y solo si:

- son independientes entre sí
- provienen de la misma distribución de probabilidad

La idea de la *muestra aleatoria* es intuitiva, se trata de un conjunto de datos observados en igualdad de condiciones y sin sesgos posibles. Básicamente los datos que tendremos recogidos en una base de datos a estudiar.

Estadísticos

Definida la muestra aleatoria, veremos la manera de estimar resultados a partir de los datos observados. Estas estimaciones se llaman *estadísticos*.

Definición 3.11. Llamaremos *estadístico* a una función de variables aleatorias observadas, que a su vez es una variable aleatoria observable y no contiene ningún parámetro desconocido.

El ejemplo más común de estadístico es la *media muestral*:

$$\mu = \bar{x} = \bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3.9}$$

Es evidente que se trata de un estadístico al estar calculado a partir de los datos observados, a diferencia de la esperanza de X que sería la verdadera media poblacional.

Otro ejemplo es la *desviación estándar*, estadístico que cuantifica la variación de una muestra:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{3.10}$$

Varianza muestral

Los estadísticos que más nos conciernen ahora mismo son aquellos que calculan las variabilidades de las muestras aleatorias, el principal de ellos es la *varianza muestral*.

Definición 3.12. Sea X una variable aleatoria con una muestra de tamaño n, $\underline{x} = (x_1, x_2, \ldots, x_n)^T$ donde x_i es la i-ésima observación de X. Definimos como *varianza muestral* al estadístico:

$$S_n^2 = S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \quad \text{para} \quad n > 1 \tag{3.11}$$

El valor que nos devuelva el estadístico define lo variable que es la muestra, a mayor valor, mayor variabilidad.

La ecuación expuesta guarda gran similitud con la fórmula de la varianza para variables aleatorias, la definición de varianza muestral proviene de sustituir las esperanzas por medias muestrales y sustituir el factor $\frac{1}{n}$ por $\frac{1}{n-1}$ para corregir el sesgo creado por estimar la esperanza de X por su media muestral, este sesgo se llama "*corrección de Bessel*".

Matriz de covarianzas muestral

Siguiendo la línea de pensamiento por la que llegamos a la definición de la varianza muestral, daremos las dos siguientes:

Definición 3.13. Sean X e Y dos variables aleatorias con muestras de tamaño n, $\underline{x} = (x_1, x_2, \ldots, x_n)^T$ e $\underline{y} = (y_1, y_2, \ldots, y_n)^T$ donde x_i e y_i son las i-ésimas observaciones de X e Y respectivamente. Definimos como *covarianza muestral* al estadístico:

$$Cov(X, Y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \quad \text{para} \quad n > 1 \tag{3.12}$$

Definición 3.14. Sean $X_1, X_2, \ldots, X_p$ p variables aleatorias de las que se recogen una muestra de tamaño n, siendo x_{ij} el valor de la j-ésima variable en la i-ésima observación, $\forall i = 1, \ldots, n$ y $\forall j = 1, \ldots, p$, y $\underline{x_i} = (x_{i1}, \ldots, x_{ip})$, $\forall i = 1, \ldots, n$, el vector de las muestras de todas las variables en la i-ésima observación. Definimos como *matriz de covarianzas muestral* al estadístico:

$$S = \frac{1}{n-1} \sum_{i=1}^{n} (\underline{x_i} - \underline{\bar{x}})(\underline{x_i} - \underline{\bar{x}})^T \quad \text{para} \quad n > 1 \tag{3.13}$$

Donde $\underline{\bar{x}} = \frac{1}{n} \sum_{i=1}^{n} \underline{x_i} \in \mathbb{R}^p$ es el vector de medias muestrales de las variables aleatorias.

Estos estadísticos estimarán la variabilidad de las muestras observadas, serán esenciales para analizar los datos y poder realizar el *análisis de componentes principales*.

3.2– Introducción a la descomposición espectral de matrices

Seguimos aclarando conceptos. Ahora dejamos la estadística para introducirnos en el álgebra, y usaremos esta sección para comentar los aspectos fundamentales de la descomposición espectral de matrices. Presentaremos brevemente las nociones de **autovalor** y **autovector**, además de un teorema relacionado.

Sea $A \in \mathbb{C}^{n \times n}$, una matriz cuyos coeficientes son números reales o complejos, podríamos entender A como una transformación lineal entre $\mathbb{C}^n$ y $\mathbb{C}^n$, tal que para cada vector $\mathbf{x} \in \mathbb{C}^n$, se tiene:

$$A : \mathbf{x} \mapsto A\mathbf{x} \tag{3.14}$$

El vector resultante de la transformación lineal, $A\mathbf{x}$, puede variar tanto en dirección como en magnitud. Si nos centramos en los casos en los que la dirección de los vectores permanece intacta, y solo varía su longitud, nos estaremos acercando a la idea detrás de los autovectores y autovalores.

Definición 3.15. Sea $A \in \mathbb{C}^{n \times n}$. Si un escalar $\lambda \in \mathbb{C}$ y un vector no nulo $\mathbf{x} \in \mathbb{C}^n$ cumplen la ecuación:

$$A\mathbf{x} = \lambda\mathbf{x} \tag{3.15}$$

llamaremos a λ *autovalor* de A y a $\mathbf{x}$ *autovector* de A asociado con λ.

El par autovector-autovalor siempre estará relacionado entre sí, a esta dupla se le denominará *par propio*.

Se deduce de la definición que al ser el autovector no nulo, el autovalor tampoco podrá serlo.

3.2.1. Teorema espectral para matrices normales

Además de las respectivas definiciones, enunciaremos el *teorema espectral para matrices normales*, resultado que usaremos posteriormente. Damos una versión reducida del mismo con lo necesario para desarrollar el *análisis de componentes principales*. En él se mencionan los conceptos de *matriz normal* y *vectores ortonormales* que definiremos a continuación.

Definición 3.16. Una matriz A se dice *normal* si y solo si $AA^* = A^*A$, siendo A^* la matriz A traspuesta y conjugada.

Evidentemente, en el caso de que la matriz tenga coeficientes reales, valdrá con que se cumpla $AA^T = A^TA$ para que la matriz sea normal. De esta forma, todas las matrices simétricas reales son normales.

Definición 3.17. Sean $v_1, \ldots, v_n$ vectores reales se dice que son vectores *ortonormales* si y solo si se cumple que:

- Tienen norma unitaria, $v_i \cdot v_i = 1$, $\forall i = 1, \ldots, n$.
- Son ortogonales, $v_i \cdot v_j = 0$, $\forall i, j = 1, \ldots, n$ con $i \neq j$.

Tras estas dos sencillas definiciones, enunciamos el teorema mencionado.

Teorema 3.1. *Sea A una matriz $n \times n$, con autovalores $\lambda_1, \ldots, \lambda_n$. Las siguientes afirmaciones son equivalentes:*

(a) A es normal.

(b) A tiene exactamente n autovectores ortonormales.

3.3– Análisis de las relaciones entre variables: Matriz de covarianza

Comenzamos con el desarrollo propio del *análisis de componentes principales*. En él buscaremos analizar las diferentes observaciones que se tienen almacenadas en una base de datos para determinar similitudes entre ellas. Utilizando las herramientas matemáticas discutidas anteriormente, buscaremos aislar las propiedades o los patrones más importantes, las que nos otorguen mayor variabilidad entre los datos disponibles, con el fin de agrupar o visualizar la información de manera más clara teniendo menor número de componentes por observación. Los nuevos patrones que asignemos a las observaciones serán nuestras **componentes principales**.

Para aplicar esta técnica de análisis, partiremos de una base de datos formada por n observaciones numéricas, donde cada observación tendrá p variables distintas, es decir, tenemos una muestra aleatoria de n vectores aleatorios de dimensión p. Notaremos el vector de las observaciones correspondientes a la j-ésima variable por $\underline{x_j} = (x_{1j}, x_{2j}, \ldots, x_{nj})^T$, $\forall j = 1, \ldots, p$, y definimos una matriz $n \times p$, X, que los contiene en sus columnas.

$$X = \left(\underline{x_1}|\underline{x_2}|\cdots|\underline{x_p}\right) = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} \tag{3.16}$$

El elemento de la matriz x_{ij} representará el i-ésimo valor de la j-ésima observación.

3.3.1. Estandarización de los datos

El primer paso para realizar un análisis entre variables es asegurarnos que se encuentran en magnitudes que las hagan comparables. En muchos casos se dispondrán de variables en escalas muy distintas y que harán imposible una justa comparación entre sus variabilidades. Las variables que posean una mayor magnitud o mayor varianza natural tendrán una influencia injustificada que vendría de la naturaleza de las variables, no de su importancia en el modelo.

Un ejemplo sería tener como variables los ingresos anuales en miles de euros y el número de hijos por familia, los ingresos tendrán un mayor valor que el número de hijos y una mayor variabilidad, el número de hijos se suele mover en los mismos valores y los ingresos entre familias pueden variar mucho, pero no por ello una variable es más importante que la otra. En caso de necesitarse, según la naturaleza de los datos, distinguiremos dos normalizaciones que realizar a una variable aleatoria:

Normalización con media y desviación estándar

Es la más recomendable, elimina los problemas de dispersión y de escalas.

$$x_{ij}^* = \frac{x_{ij} - \mu_j}{\sigma_j} \tag{3.17}$$

Siendo μ_j la media muestral y σ_j la desviación estándar de $\underline{x_j}$.

Centrado de datos

Es otro tipo de normalización donde no se estandariza la variabilidad entre variables, simplemente centra los datos alrededor del cero. Es útil en casos en los que las variables están en la misma escala.

$$x_{ij}^* = x_{ij} - \mu_j \tag{3.18}$$

Siendo μ_j la media muestral de $\underline{x_j}$.

Denotaremos por X^* a la matriz X normalizada con valores x_{ij}^*.

3.3.2. Matriz de covarianza

Con los datos ya normalizados será sencillo obtener la matriz de covarianza. Recordando la fórmula de esta:

$$S = \frac{1}{n-1}\sum_{i=1}^{n}(\underline{x_i} - \underline{\bar{x}})(\underline{x_i} - \underline{\bar{x}})^T = \frac{1}{n-1}(X - \mathbf{1}_n \cdot \underline{\bar{x}})^T(X - \mathbf{1}_n \cdot \underline{\bar{x}}) \quad \text{para} \quad n > 1 \tag{3.19}$$

Siendo $\mathbf{1}_n$ el vector de unos de dimensión $1 \times n$.

Podemos observar que $(\underline{x_i} - \underline{\bar{x}})$ no es más que el proceso de centrar los datos, algo que ya hemos realizado en la normalización, por lo tanto podemos expresar la matriz de covarianza en la siguiente fórmula matricial:

$$S = \frac{1}{n-1}(X^*)^T X^* \tag{3.20}$$

3.4– Componentes principales

Después del análisis de las variables iniciales, el siguiente paso es erigir unas nuevas más importantes o significativas. Para ello vamos a buscar las combinaciones de vectores columna de X, representando a las variables originales, con mayor variabilidad, es decir, aquellas que maximicen la varianza. A estos patrones o combinaciones de variables los llamaremos **componentes principales** y serán las variables que asignaremos al final del proceso.

3.4.1. Determinación de las componentes principales

Como ya se mencionó, buscaremos aquellas combinaciones lineales de columnas de X con mayor variabilidad. Siendo $\underline{a} = (a_1, \ldots, a_p)$ un vector de constantes cualquiera, sabemos que las combinaciones lineales tendrán la forma:

$$\sum_{j=1}^{p} a_j \cdot x_j = X\underline{a} \tag{3.21}$$

Por medio de la matriz de covarianza S de X, podemos expresar la análoga de $X\underline{a}$:

$$Var(X\underline{a}) = \underline{a}^T Var(X)\underline{a} = \underline{a}^T S\underline{a} \tag{3.22}$$

Para tener una solución bien definida para este problema de maximizar la varianza, nos restringiremos a los vectores $\underline{a}$ unitarios, es decir, los que cumplan $\underline{a}^T\underline{a} = 1$. El problema a maximizar quedaría con la restricción de la siguiente manera:

$$\max_{\underline{a}\in\mathbb{R}^p} \underline{a}^T S\underline{a} - \lambda(\underline{a}^T\underline{a} - 1) \tag{3.23}$$

Encontramos el máximo a partir de la función Lagrangiana:

$$\mathcal{L}(\underline{a}, \lambda) = \underline{a}^T S\underline{a} - \lambda(\underline{a}^T\underline{a} - 1) \tag{3.24}$$

La cual derivamos con respecto a $\underline{a}$ e igualamos a cero:

$$\frac{\partial \mathcal{L}}{\partial \underline{a}} = 2S\underline{a} - 2\lambda\underline{a} = 0 \Rightarrow S\underline{a} = \lambda\underline{a} \tag{3.25}$$

En este punto, reconocemos la ecuación $S\underline{a} = \lambda\underline{a}$ como propia de la descomposición en autovalores y autovectores, haciendo de $\underline{a}$ un autovector unitario de S y de λ su autovalor. Sustituyendo en la fórmula de la varianza, tenemos que la varianza máxima posible es:

$$Var(X \cdot \underline{a}) = \underline{a}^T S\underline{a} = \lambda\underline{a}^T\underline{a} = \lambda \tag{3.26}$$

Obtenemos como conclusión, la combinación lineal de columnas de X con mayor varianza es aquella de la forma $X\underline{a}$ donde $\underline{a}$ es el autovector unitario con el autovalor más alto. Por lo tanto, $X\underline{a}$ será la **primera componente principal**.

Propiedad de incorrelación entre componentes principales

Toda matriz real simétrica cuadrada es normal, por lo que podemos aplicar el **teorema espectral para matrices normales** sobre S. Tendremos que S tiene exactamente p autovalores reales y que los autovectores pueden expresarse de forma **ortonormal**.

Denotando por $\lambda_1, \ldots, \lambda_p$ los autovalores de S ordenados de mayor a menor, es decir, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p$, y $\underline{a_1}, \ldots, \underline{a_p}$ sus respectivos autovectores unitarios ortonormales entre sí. Al ser ortonormales, tendremos que los vectores no tendrán una dirección común en la que coincidan, es decir, $\underline{a_k} \cdot \underline{a_{k'}} = 0$, $\forall k \neq k'$. Esto hará que la covarianza de cada par de autovectores distintos sea nula, los autovectores serán **incorrelados**.

Debido a la incorrelación de los autovectores, $\underline{a_k}$, $\forall k = 1, \ldots, p$, tenemos que las combinaciones lineales de X determinadas por ellos mismos, $X\underline{a_k}$, $\forall k = 1, \ldots, p$, también serán incorreladas. Esta incorrelación implica que cada una de las combinaciones lineales descritas expliquen una porción de la varianza que no es explicada por las demás. Teniendo así que para $k = 1, \ldots, p$ la combinación lineal de X con mayor varianza no explicada por $X\underline{a_1}, \ldots, X\underline{a_{k-1}}$ sea $X\underline{a_k}$. Definiendo así la **k-ésima componente principal** como $X\underline{a_k}$, $\forall k = 1, \ldots, p$.

3.4.2. Criterios para la selección de componentes principales

Dado que las componentes principales son incorreladas, cada una explica una parte de la varianza en exclusiva, cuantificada por su autovalor asociado, las p componentes explicarán la totalidad de la varianza de la muestra, y esta podrá ser calculada mediante la suma de las varianzas individuales, no existirá redundancia.

Al porcentaje de la varianza total que explique cada componente principal lo llamaremos **varianza explicada**. Al depender exclusivamente del tamaño del autovalor asociado, las calcularemos en base a estos y denotándolas con π_j, $\forall j = 1, \ldots, p$.

$$\pi_j = \frac{\lambda_j}{\sum_{j=1}^{p} \lambda_j} = \frac{\lambda_j}{traza(S)} \tag{3.27}$$

También es importante el concepto de *varianza explicada acumulada*, nos dará una idea de cuántas componentes principales debemos de seleccionar en cada caso. Será la suma de las k primeras varianzas explicadas: $\sum_{i=j}^{k} \pi_j$, $\forall k = 1, \ldots, p$.

El número de componentes principales que deberían ser elegidas para formar parte del análisis es indefinido, en la mayoría de los casos se recomendará el número necesario para que se disponga de una varianza explicada acumulada en el modelo al menos del 80 %. El umbral aconsejable es entre el 80 % y 90 %, por debajo nos quedaríamos sin una parte sustancial de las componentes de la muestra y por arriba empezarían a entrar variables menos importantes no significativas, se las conoce como **ruido**. Al tener poca importancia suelen estar dominadas por la aleatoriedad y añadir más variables harán el modelo nuevo menos interpretable.

Sin embargo, en algunos casos se preferirá disponer de las dos o tres primeras componentes principales que permiten una fácil visualización gráfica, aunque eso reduzca la varianza explicada acumulada a un valor inferior al deseado, siempre que no caiga por debajo del 70 %, lo que significaría una perdida excesiva de la información.

3.4.3. Proyección de los datos en el espacio de componentes principales

El paso final en el *análisis de componentes principales*, es el de conseguir los datos de la muestra en función de las nuevas componentes principales. El proceso a seguir será la proyección de las n observaciones que se encontrarán en el espacio de p variables inicial sobre el nuevo espacio con un número $q < p$ elegido de componentes iniciales. La proyección se ejecutará con la matriz A_q de dimensión $p \times q$ cuyas columnas son los autovectores correspondientes a las q primeras componentes iniciales, $\underline{a_k}$, $\forall k = 1, \ldots, q$.

$$A_q = \left(\underline{a_1}|\underline{a_2}|\cdots|\underline{a_q}\right) \tag{3.28}$$

Siendo X^* la matriz de dimensión $n \times p$ de los datos normalizados, la proyección se realiza por medio del producto de ambas matrices. Multiplicando obtenemos una proyección de cada observación, ubicadas en las filas de X^*, sobre las nuevas componentes principales. El resultado tendrá forma de matriz $n \times q$ y será una representación de los datos en el nuevo subespacio de las componentes principales.

3.5– Interpretación de los resultados

Finalizado el *análisis de componentes principales*, habremos conseguido trasladar gran parte de la información que existía de n observaciones en un espacio de p parámetros a un espacio ortogonal de $q < p$, donde esos q parámetros definen las q mayores diferencias entre los datos, siendo totalmente independientes unas de otras.

Los usos prácticos de esta técnica son abundantes. Para facilitar la comprensión del mecanismo de uso, daremos un breve ejemplo, además del que viene desarrollado en el apartado de "**Aplicaciones**" de este trabajo.

Si obtenemos una serie de datos de una gran cantidad de deportistas de élite de diversos deportes, entre los que se encuentran más de 200 datos entre biométricos como pueden ser altura, peso, diámetro del muslo, porcentaje graso, envergadura, capacidad aeróbica, etc.; y resultados de pruebas físicas como pueden ser tiempos en carrera, frecuencia cardiaca en reposo o pesos levantados en ejercicios de fuerza; no podremos sacar grandes conclusiones de primeras, los datos serán difíciles de entender a simple vista por su inmensidad. Además muchos de los datos estarán altamente relacionados, el atleta más alto también será el que tenga mayor envergadura, y el más pesado el que tenga mayor diámetro a la altura del pecho. Sin embargo, realizando las técnicas descritas, obtendremos unas nuevas variables que darán un valor a cada deportista. Estas variables serán las **componentes principales** y vendrán determinadas por las mayores variaciones. En este caso, la primera componente probablemente daría un alto valor a los jugadores de baloncesto o rugby, y bajo a los gimnastas o maratonianos, ésta organizaría los deportistas según su tamaño, tanto a lo ancho como a lo largo, acumulando la información de todas las variables que guardan relación de este tipo, en general el peso y las medidas. Una segunda componente podría diferenciar a los deportistas según lo aeróbico que sea su deporte, diferenciará los ciclistas y atletas, de halterófilos o lanzadores, esta relacionaría otras variables como tiempos en carrera de larga distancia o la frecuencia cardiaca. Con estas dos primeras ya podríamos exponer los resultados en una gráfica en dos dimensiones con cada eje siendo la puntuación en cada componente principal. Si se quiere mayor riqueza en los datos obtenidos, debido a la gran cantidad de datos recogidos, sería más inteligente profundizar más en el análisis

sacando más componentes. Algunas otras irán acorde a la potencia explosiva, la agilidad o la flexibilidad. Teniendo así un análisis mucho más claro e intuitivo que el inicial.

Los beneficios de realizar el análisis son más que evidentes en todo lo relacionado con la interpretabilidad de los datos.

CAPÍTULO 4

Implementaciones

En este capitulo desarrollaremos las implementaciones de las técnicas de *machine learning* presentadas anteriormente. Armaremos en *Haskell*, un lenguaje de programación funcional, un **clasificador** capaz de asignar clases predefinidas a cadenas de texto basándonos en el *clasificador de Naive Bayes*, y un **recomendador** que a partir de un fragmento de texto concreto utilizará el *análisis de componentes principales* para darnos los textos de una base de datos que mayor similitud guarden con el primero.

4.1– Lenguaje *Haskell*

Haskell es un lenguaje de programación puramente funcional, creado en 1990, y cuyo nombre hace referencia al matemático estadounidense Haskell Brooks Curry, pieza fundamental en el desarrollo de los lenguajes de programación funcionales.

La programación funcional está fundamentada en el uso de funciones puras, siendo estas aquellas cuyo valor de retorno está solamente determinado por los valores de entrada, las funciones toman un valor principal en el lenguaje. Así como también es un lenguaje de operación perezosa, sólo realiza un cálculo cuando este es estrictamente necesario y hace falta para continuar, esto puede ser interesante a la hora de optimizar el rendimiento del programa, siempre que se tenga cuidado con las fugas de memoria que pueda generar.

El uso de lenguajes de programación funcional y evaluación perezosa, siendo *Haskell* el más conocido y usado entre ellos, es de especial interés en el mundo de las matemáticas por su similitud con el lenguaje matemático.

4.2– Implementación: Clasificador Naive Bayes

Para hacer funcionar el clasificador, partiremos de una base de datos con texto organizado en distintas categorías. La misión del clasificador será asignar una categoría concreta de la base de datos a una cadena de texto que aportemos.

Ejemplos de uso incluyen valorar si un comentario es negativo o positivo, si un mensaje es *spam* o no, o, como haremos en el próximo capítulo, determinar el idioma de un fragmento de texto.

La elaboración de la base de datos es una parte crucial para crear un clasificador Naive Bayes. En el próximo capítulo veremos un ejemplo de ello. Para esta implementación,

consideramos que tenemos una base de datos, a la que nos referiremos como `datos`, con estructura `[(String,String)]`. Los pares tendrán como primera componente un fragmento de texto a estudiar y como segunda la categoría a la que pertenecen. Una vez terminada la aplicación, el texto que aportaremos como entrada a clasificar, se le asignará una de estas categorías.

Con los datos organizados, podemos comenzar con el desarrollo de la librería. Nosotros crearemos un nuevo archivo `Clasificadors.hs` donde ubicaremos el clasificador.

4.2.1. Declaración del módulo e importación de librerías

Creamos y declaramos un nuevo módulo llamado `Clasificador`, sobre él implementaremos el clasificador:

Código 4.1: Declaración del módulo `Clasificador`

```
1 module Clasificador where
```

Código 4.1: Declaración del módulo `Clasificador`

En el proceso, necesitaremos la importación de las siguientes librerías:

- `Data.List`, librería disponible por defecto en el entorno de haskell que usaremos para operar con listas, la importamos explícitamente para poder usar sus funciones con un alias, `L`, y confrontar posibles errores de ambigüedad.
- `Data.Hashable` nos ayuda a crear datos con estructura `HashMap`, implementados desde la librería `Data.HashMap.Strict`. Los datos de estructura `HashMap` nos ayudarán con el manejo de los datos ya que nos aportan facilidades a la hora de realizar busquedas entre los mismos.
- `Data.Ord` nos proporciona funcionalidades relacionadas con el orden de los elementos que nos serán de utilidad.

Código 4.2: Librerías usadas en `Clasificador`

```
1 import Data.List as L
2 import Data.Hashable
3 import Data.HashMap.Strict as HM
4 import Data.Ord
```

Código 4.2: Librerías usadas en `Clasificador`

4.2.2. Función `frecuencia`

Definimos una función auxiliar que utilizaremos en diversas ocasiones. Su utilidad será la de contar los elementos de una lista integrándolos en un `HashMap`. Esta estructura está formada por pares de elementos donde el primero, la clave, será un elemento de la lista a contar y el segundo, el valor, el número de veces que se repite el elemento en la lista, en otras palabras la frecuencia de veces que aparece, de ahí el nombre de la función, `frecuencia`.

Código 4.3: Función `frecuencia`

```
frecuencia :: (Eq k, Data.Hashable.Hashable k, Integral v) => [k] -> HashMap k v
frecuencia [] = HM.empty
frecuencia (x:xs) = HM.insertWith (+) x 1 (frecuencia xs)
```

Código 4.3: Función `frecuencia`

4.2.3. Funciones para la probabilidad por clase

La primera probabilidad a calcular para aplicar el algoritmo de Naive Bayes es la probabilidad a priori, en este caso, estamos hablando de la probabilidad de que el texto pertenezca a una clase u otra sin tener información previa al respecto.

La función `frecClase` producirá un `HashMap` donde la clave representa cada clase y el valor es el número de veces que se repite en la base de datos, es decir, el número de entradas que existen que pertenecen a esa categoría.

Código 4.4: `HashMap frecClase`

```
frecClase :: [(String, String)] -> HashMap String Double
frecClase datos =
  HM.map fromIntegral (frecuencia (L.map fst datos))
```

Código 4.4: `HashMap frecClase`

La función de probabilidad queda implementada en nuestro módulo con `probClase`. El número de extractos, `nExtractos`, no es más que la suma de todas las entradas o extractos de la base de datos, la cantidad de elementos sin repetición. El valor de ese número lo introduciremos directamente en la función, ya que no variará por idioma y nos ahorraremos cálculos no teniendo que calcular la suma para cada idioma, haciendo la aplicación más eficiente.

Código 4.5: Función `probClase`

```
probClase :: String -> Double -> HashMap String Double -> Double
probClase clase nExtractos frecuencias =
  lookupDefault 0 clase (HM.map (/nExtractos) frecuencias)
```

Código 4.5: Función `probClase`

4.2.4. Funciones para la probabilidad por palabra dada la clase

Tomando como referencia las anteriores funciones de la probabilidad de la clase, el cálculo de la probabilidad de cada palabra dada la clase toma el mismo patrón.

Tenemos una función `frecPalabraPorClase` que nos devolverá las frecuencias de las palabras de una clase concreta. Lo primero será mirar hacia la secuencia "`L.filter (\x ->clase == (snd x)) datos`", que consistiría en una lista que filtra los elementos de `datos` quedándose con aquellos de la clase elegida. Se realiza un `L.map` sobre esta lista para operar sobre cada elemento de la misma, se filtran los extractos de texto de ese tipo, se extraen y se separan por palabras, creando una lista con todas las palabras de esa clase. A esa lista de palabras se le aplica `frecuencia` para obtener el número de veces que se repite cada palabra en la clase, todo ello organizado en un `HashMap`. Los elementos del

`HashMap` estarían conformados por una palabra en la clave y el número de veces que se repite el valor en la clase.

Código 4.6: Función `frecPalabraPorClase`

```
frecPalabrasPorClase :: [(String, String)] -> String -> HashMap String Integer
frecPalabrasPorClase datos clase =
   frecuencia (concatMap words (L.map snd
              (L.filter (\x-> clase == (fst x)) datos)))
```

Código 4.6: Función `frecPalabraPorClase`

La siguiente función calcula la probabilidad de que dadas una palabra y una clase, entre todas las palabras de esa clase, se escoja al azar una palabra y sea la dada. Esta probabilidad estará suavizada por el **suavizado de Laplace (Laplace Smoothing)**, como se explicó en la parte teórica, de esta manera, sortearemos el **Problema de Cero Frecuencia**.

`probSuaviPalabraDadaClase` almacena en `frecPalabrasEnClase` el `HashMap` de palabras y sus frecuencias de la clase especificada. En `nPalabrasEnClase` se guarda el número de elementos de `frecPalabrasEnClase`, es decir, el número de palabras de esa clase, y en `nPalabraEnClase` la frecuencia de la palabra elegida. Posteriormente, se calcula la probabilidad suavizada por medio de la fórmula presentada en la parte teórica:

$$P'(B_i \mid A) = \frac{nAB_i + \alpha}{nA + \alpha \cdot N} \tag{4.1}$$

Tomando $\alpha = 1$, por simplicidad.

Código 4.7: Función `probSuaviPalabraDadaClase`

```
probSuaviPalabraDadaClase :: String -> String
                        -> [(String, String)] -> Double -> Double
probSuaviPalabraDadaClase palabra clase datos nPalabras =
   let frecPalabrasEnClase = frecPalabrasPorClase datos clase
       nPalabrasEnClase = fromInteger $ sum $ HM.elems frecPalabrasEnClase
       nPalabraEnClase = fromInteger $
                         lookupDefault 0 palabra frecPalabrasEnClase
   in (nPalabraEnClase + 1)/(nPalabrasEnClase + nPalabras)
```

Código 4.7: Función `probSuaviPalabraDadaClase`

4.2.5. Función para la verosimilitud de la clase dado un texto

Llegamos a la parte final de la construcción del clasificador, nos queda completar la fórmula donde se respaldará el clasificador para calcular la verosimilitud de que, dado un fragmento de texto, este pertenezca a una clase u otra. Con ayuda de las funciones `probSuaviPalabraDadaClase` y `probClase`, implementamos el resultado de:

$$P(A) \cdot \prod_{i=1}^{n} P'(B_i \mid A) \tag{4.2}$$

Llamaremos a la nueva función `verosiClaseDadoTexto`. Destacar que el producto se hace creando una lista con `L.map` iterando sobre las palabras del texto de la lista `(words`

`texto`), y que el resto de las variables se usarán ya calculadas para mejorar la eficiencia, no dependen de la clase por lo que sería un gasto innecesario calcularlas para cada clase.

Código 4.8: Función `verosiClaseDadoTexto`

```
verosiClaseDadoTexto :: String -> String -> Double ->
       HashMap String Double -> [(String, String)]-> Double -> Double
verosiClaseDadoTexto clase texto nPalabras frecuencias datos nExtractos =
    let pTexto = L.map (\palabras -> probSuaviPalabraDadaClase palabras clase
                                     datos nPalabras) (words texto)
        pClase = probClase clase nExtractos frecuencias
    in pClase * (product pTexto)
```

Código 4.8: Función `verosiClaseDadoTexto`

4.2.6. Clasificadores

Presentaremos dos versiones del clasificador, la primera será `clasificadorClases` que básicamente consiste en evaluar la función anterior, `verosiClaseDadoTexto`, para todas las clases. Nos devolverá una lista formada por pares donde el primer elemento representará una clase y el segundo elemento la verosimilitud de que el texto dado sea de cada clase. La clase que tenga un valor de verosimilitud mayor será la que se le asignará.

Esta versión necesitará de los siguientes parámetros:

- `texto`: el fragmento de texto a clasificar en formato `String`.
- `datos`: los datos en formato `[(String,String)]`, donde la primera componente del par es un fragmento de texto del conjunto de entrenamiento y la segunda componente representa la clase a la que se asocia ese fragmento.
- `frecuencias`: la frecuencia con la que aparece cada clase en `datos` en formato `HashMap String Double` donde la clave representa cada clase valor el número de elementos en `datos` de esa clase.
- `clases`: una lista cuyos elementos representan las clases en formato `[String]`.
- `nPalabras`: el número de palabras sin repetición que aparecen en los fragmentos de texto de `datos` en formato `Double`.
- `nExtractos`: el número de elementos de `datos` en formato `Double`.

Código 4.9: Función `clasificadorClases`

```
clasificadorClases :: String -> [(String,String)] -> HashMap String Double
        -> [String] -> Double -> Double -> [(String, Double)]
clasificadorClases texto datos frecuencias clases nPalabras nExtractos =
 L.map (\clase ->(clase, verosiClaseDadoTexto clase texto nPalabras
                                  frecuencias datos nExtractos)) clases
```

Código 4.9: Función `clasificadorClases`

Esta última función que hemos presentado es la versión explícita ya que se espera que el usuario proporcione los parámetros asociados a la base de datos de manera externa. La siguiente es la versión implícita o autoajustada de la misma, `clasificador`. En este caso, los parámetros serán calculados por la propia función y no hará falta más que el texto a clasificar y la base de datos en los formatos mencionados con anterioridad. Se calcularán `frecuencias`, `clases`, `nPalabras` y `nExtractos` a partir de `datos`. Se evaluará la función explícita anterior, `clasificadorClases`, pero sólo se devolverá un `String` que representará a la clase con mayor verosimilitud de ser la asociada al fragmento de texto dado.

Código 4.10: Función `clasificador`

```
clasificador :: String -> [(String,String)] -> String
clasificador texto datos =
    let frecuencias = frecClase datos
        clases = HM.keys $ (frecuencia . L.map fst) datos
        nPalabras = fromInteger $ sum $ HM.elems
                    ((frecuencia . concatMap words) (L.map snd datos))
        nExtractos = sum $ HM.elems frecuencias
        probs = clasificadorClases texto datos frecuencias
                                   clases nPalabras nExtractos
    in fst (L.maximumBy (comparing snd) probs)
```

Código 4.10: Función `clasificador`

4.3– Implementación: Recomendador

La implementación del *análisis de componentes principales* que presentaremos en este capítulo se basa en una función que toma una base de datos donde se hallan fragmentos de texto junto a un título, nombre o identificador que los distingan y, por medio de las técnicas del *análisis de componentes principales*, proporcione una lista con los identificadores de los textos de la base de datos que guarden mayor similitud con un elemento dado de la base de datos.

Los ejemplos de uso de esta aplicación pueden ser muy diversos. Por nombrar algunos, si cada fragmento de texto es una enumeración de películas que le gustó a una persona y el identificador se refiere a la persona, introduciendo una serie de películas que le gustan a una persona concreta, la aplicación ordenará las personas de la base de datos con gustos más similares a los de la persona dada. Sin salir del mundo cinematográfico, otro ejemplo puede ser tener como texto una sinópsis de una película, la aplicación será capaz de dar las películas más similares a una dada. Más adelante, en el próximo capítulo se dará un ejemplo de aplicación que devuelve artículos de la enciclopedia *Wikipedia* similares a uno dado.

Como en la implementación anterior, partiremos de una base de datos dada, que nombraremos `datos`, con formato `[(String,String)]`, donde el primer elemento del par será el nombre o título del texto que se encuentra en el segundo elemento del par. Crearemos una base de estas características en la aplicación a la que ya se hizo referencia.

4.3.1. Declaración del módulo e importación de librerías

Ubicaremos todo el contenido de la librería en un archivo llamado `Recomendador.hs`, la primera línea irá dedicada a la declaración del módulo homónimo.

Código 4.11: Declaración del módulo `Recomendador`

```
1 module Recomendador where
```

Código 4.11: Declaración del módulo `Recomendador`

Para la elaboración de la implementación, necesitaremos herramientas de las librerías que importaremos a continuación.

Entre ellas podemos reconocer `Data.List`, `Data.Hashable`, `Data.HashMap.Strict` y `Data.Ord`, librerías importadas en el anterior problema. Recordaremos brevemente sus características, además de presentar el resto.

- `Data.List` es una librería sobre listas incluida por defecto que importamos con un alias para evitar posibles problemas de ambigüedad.
- `Data.Hashable` y `Data.HashMap.Strict` nos permitirán usar el formato `hashMap` y sus útiles funciones específicas.
- `Data.Ord` aporta funciones relacionadas con el orden de elementos.
- `Numeric.LinealAlgebra.Data` y `Numeric.LinearAlgebra.HMatrix` permiten la utilización del tipo de dato `Matrix` con forma de matriz, además de herramientas para hallar los autovalores o autovectores de una matriz y otras utilidades relacionadas con las matrices.
- `Data.Maybe` importará el tipo de dato de estructura `Maybe`, basado en un dato que puede existir o no, en caso de no existir devolverá un `Nothing`, y en caso contrario `Just` seguido del dato. De esta librería utilizaremos dos funciones, `elemIndex` que devuelve la posición de un dato en una lista, y en caso de que no esté el dato en la lista un `Nothing` y `fromMaybe` que da un valor predeterminado en caso de que se dé un `Nothing`.

Código 4.12: Librerías usadas en `Recomendador`

```
1 import Data.List as L
2 import Data.Hashable
3 import Data.HashMap.Strict as HM
4 import Data.Ord
5 import Numeric.LinearAlgebra.Data
6 import Numeric.LinearAlgebra.HMatrix as LA
7 import Data.Maybe
```

Código 4.12: Librerías usadas en `Recomendador`

4.3.2. Funciones para determinar las palabras más frecuentes

La primera función que definiremos ya fue explicada en la implementación del clasificador. Su funcionamiento es sencillo, a partir de una lista de datos, devuelve un `HashMap` formado por cada elemento de la lista inicial y el número de veces que se repite la palabra en esa lista.

Código 4.13: Función `frecuencia`

```
frecuencia :: (Eq k, Data.Hashable.Hashable k, Integral v) => [k] -> HashMap k v
frecuencia [] = HM.empty
frecuencia (x:xs) = HM.insertWith (+) x 1 (frecuencia xs)
```

Código 4.13: Función `frecuencia`

Para nuestro análisis muchas de las palabras no serán significativas. Existen multitud de palabras en cualquier idioma que no nos darán información, simplemente son palabras que se usan mucho en ese idioma. En español lo son los determinantes, las conjunciones y algunas palabras muy comunes, estas muchas veces dependerá del contenido de los textos a analizar. En nuestra búsqueda de similitudes entre fragmentos de texto, estas palabras serán un incordio pues no nos darán información sobre si un fragmento se parece a otro. Este tipo de palabras se llama **palabras vacías**, o en inglés "**stop words**".

Por lo tanto, necesitaremos una lista de *palabras vacías* que llamaremos `palabrasVacias`. Usaremos la anterior función `frecuencia` para obtener las palabras más repetidas en `datos`, las cuales filtraremos para quedarnos sólo con aquellas que no sean vacías. Posteriormente restringiremos la lista a las primeras `nClaves` palabras no vacías con mayor frecuencia, eliminando muchas palabras que no tienen relevancia real que aunque no sean vacías no saldrán a penas. Este es el resultado que devolverá la función `palabrasClave`, el formato será de `[(String,Integer)]`, donde el `String` hará referencia a la palabra y el `Integer` al número de veces que aparece.

El número de palabras claves elegidas, `nClaves`, es arbitrario y dependerá de la naturaleza de la base de datos. Se recomendará que su valor oscile entre las 20 y 100 palabras en la mayoría de los casos, pero dependerá mucho de los datos. Se debe cambiar y ajustar para que no entren en el análisis palabras con poca frecuencia.

Código 4.14: Función `palabrasClave`

```
palabrasClave :: [(String,String)] -> [String] -> Int -> [(String,Integer)]
palabrasClave datos palabrasVacias nClaves =
  let frecPalabras = frecuencia (concatMap words (L.map snd datos))
      palabrasNoVacias = L.filter (\(word, _) -> notElem word palabrasVacias)
                                  (HM.toList frecPalabras)
  in take nClaves (sortBy (\(_,c1) (_,c2) -> compare c2 c1) palabrasNoVacias)
```

Código 4.14: Función `palabrasClave`

La siguiente función tendrá por objetivo devolver la matriz de los datos sobre la que realizaremos el *análisis de componentes principales*, en formato `[[Double]]`. Esta matriz tendrá las frecuencias de las denominadas palabras clave. La i-ésima fila hará referencia a el i-ésimo nombre de los fragmentos de texto de la base de datos, habrá tantas filas como nombres en `nombres`, la j-ésima columna por otra parte, hará lo propio con cada una de las palabras claves elegidas anteriormente.

Código 4.15: Función `palabrasMatriz`

```
palabrasMatriz :: [(String,String)] -> [String] -> [String] -> [[Double]]
palabrasMatriz datos claves nombres =
  let frecPalabrasNombre = HM.fromList
        (L.map (\(nombre,intro) -> (nombre, frecuencia $ words intro)) datos)
  in L.map (\nombre -> L.map
          (\palabra -> frecPalabraPorNombre nombre palabra frecPalabrasNombre)
          claves) nombres
```

Código 4.15: Función `palabrasMatriz`

Para definir la anterior función, necesitaremos la siguiente función auxiliar `frecPalabraPorNombre` que calculará la frecuencia de cada palabra para un nombre concreto, sumando las frecuencias de todos los textos que se indentifiquen con un mismo nombre.

Código 4.16: Función `frecPalabraPorNombre`

```
frecPalabraPorNombre :: String -> String ->
                        HashMap String (HashMap String Integer) -> Double
frecPalabraPorNombre nombre palabra frecuencias =
   fromIntegral $ HM.lookupDefault 0 palabra (frecuencias HM.! nombre)
```

Código 4.16: Función `frecPalabraPorNombre`

4.3.3. Análisis de componentes principales

Teniendo ya las frecuencias organizadas, pasaremos a realizar el *análisis de componentes principales*, con la idea de aplicar estas funciones a la función formada anteriormente.

La primera función devolverá los autovalores y autovectores de la matriz de covarianzas de una matriz dada. Haremos estos cálculos de manera muy sencilla con las funciones de la librería importada `Numeric.LinearAlgebra.HMatrix` como son `meanCov` para la matriz de covarianzas, o `eigSH` para la descomposición espectral.

Código 4.17: Función `eigDescomposicion`

```
eigDescomposicion :: [[Double]] -> (Vector Double, Matrix Double)
eigDescomposicion matriz =
    let vectores = fromLists matriz
        (_, covMatriz) = meanCov vectores
        (autovalores, autovectores) = eigSH covMatriz
    in (autovalores, autovectores)
```

Código 4.17: Función `eigDescomposicion`

Otro de los problemas que comentamos en la sección teórica es la elección del número de autovalores, que se convertirá en el número de componentes principales. Para resolver esta duda tendremos la función `nAutovalores` que en función del vector de los autovalores y un porcentaje mínimo de varianza explicada expresado en tanto por 1, da como respuesta un valor de tipo `Int` que es el número de autovalores que explicarán ese porcentaje dado de varianza.

La función calcula la varianza explicada acumulada de añadir cada uno de los autovalores y se queda con el primero de los valores que cumpla la condición de que sea mayor que

el porcentaje dado. En el caso de que todos los valores de la varianza acumulada sean menores que el porcentaje dado, se devolverá el número total de autovalores.

Código 4.18: Función `nAutovectores`

```
nAutovectores :: Vector Double -> Double -> Int
nAutovectores autovalores porcentaje =
    let varianzaTotal = sumElements autovalores
        varianzaExplicada = LA.toList $ cmap (/varianzaTotal) autovalores
        varianzaAcumulada = scanl1 (+) varianzaExplicada
    in fromMaybe (1) (L.findIndex (> porcentaje) varianzaAcumulada)
```

Código 4.18: Función `nAutovectores`

Terminamos el *análisis de componentes principales* con la función homónima, que recibe los autovectores en forma de `Matrix Double`, la matriz de las frecuencias `matriz` y un valor `Int`, `nAutoV`, que será el resultado de `nAutovalores`, el número de autovalores deseado.

La función `analisisComponentesPrincipales` pasará `matriz` a un formato `Matrix` con la función `fromLists`, para así poder multiplicarla con la matriz formada por los primeros `nAutoV` autovectores. Haciendo esta proyección, ayudándonos en el proceso de `tr` que transponerá las matrices para poder efectuar el producto con las debidas dimensiones, obtenemos la matriz de proyecciones principales, resultado del análisis.

Código 4.19: Función `analisisComponentesPrincipales`

```
analisisComponentesPrincipales :: Matrix Double -> [[Double]]
                               -> Int -> Matrix Double
analisisComponentesPrincipales autovectores matriz nAutoV =
  tr $ mul (tr autovectoresClave) (tr vectores)
  where
    vectores = fromLists matriz
    autovectoresClave = takeColumns nAutoV autovectores
```

Código 4.19: Función `analisisComponentesPrincipales`

4.3.4. Funciones auxiliares del recomendador

El último paso que nos queda es montar el recomendador, para ello necesitaremos dos funciones auxiliares que definiremos a continuación. La primera, `distEuclidCuadrPorFila` calculará las distancias euclídeas de todas las filas de la matriz con respecto a una concreta de ellas. Esto lo hace creando en `matrizAux` una matriz de la dimensión de `matriz` donde todas las filas son la fila número `fila` de `matriz`. Posteriormente, las resta con la matriz original en `difMatriz`, obteniendo la matriz de distancias por elemento, eleva al cuadrado elemento a elemento con `difCuadradoMatriz` y por último, suma los resultados de las columnas y devuelve un `Vector Double` cuyos elementos son la distancia de cada fila a la fila dada.

Código 4.20: Función `distEuclidCuadrPorFila`

```
distEuclidCuadrPorFila :: (Matrix Double) -> Int -> Vector Double
distEuclidCuadrPorFila matriz fila =
  sum $ toColumns difCuaradoMatriz
  where
```

```
5     d = cols matriz
6     matrizAux = repmat (subMatrix (fromIntegral fila, 0) (1, d) matriz)
7                        (rows matriz) 1
8     difMatriz = matriz - matrizAux
9     difCuaradoMatriz = difMatriz * difMatriz
```

Código 4.20: Función `distEuclidCuadrPorFila`

La otra función auxiliar, `indicesOrdenados`, es más sencilla. Simplemente devuelve la posición que tienen los elementos de `lista`, una lista con formato `[Double]`, si estos se ordenaran de menor a mayor. La idea es guardar en `listaOrdenada` la lista de valores de `lista` pero ordenados y sin repetición con `sort` y `nub` respectivamente, y a partir de esta lista ir sacando las posiciones a cada elemento de `listaOrdenada` y devolver una lista de datos `Int` con todas las posiciones en orden ascendente.

Código 4.21: Función `orden`

```
1 indicesOrdenados :: [Double] -> [Int]
2 indicesOrdenados lista =
3   let listaOrdenada = nub $ sort lista
4   in concatMap (\x -> L.elemIndices x lista) listaOrdenada
```

Código 4.21: Función `orden`

4.3.5. Recomendador

Uniremos todo el proceso anterior en la función final `recomendador`. Esta necesitará de los siguientes parámetros.

- `datos`: una lista en formato `[(String,String)]` donde el primer elemento del par es el nombre con el que se identifica el texto del segundo elemento.
- `palabrasVacias`: una lista de palabras vacías o "stop words" en formato `[String]`.
- `nombre`: el nombre o título que identifica el elemento de `datos` del que se quieren buscar recomendaciones.
- `n`: el número de recomendaciones que se quieran recibir, en formato `Int`.
- `nClaves`: el número `Int` de palabras clave que se quieran extraer de la base de datos sobre las que se aplicará el *análisis de componentes principales* para obtener posteriormente las componentes principales y las recomendaciones.
- `porcentaje`: el porcentaje deseado mínimo de la varianza explicada en el modelo, en formato `Double`.

La función evalúa de manera secuencial los resultados de las funciones anteriores, asignando los valores correspondientes a distintas variables locales mediante la construcción `let`. Haremos un repaso a todas ellas.

- `claves`: se queda con los `nClaves` primeros valores de los pares de la lista de pares `palabrasClave`, es decir, con las palabras clave sin su frecuencia, es una lista de `String`.

- `nombres`: una lista de `String` con los nombres/títulos de los fragmentos de texto de `datos` sin repetición.
- `matriz`: simplemente el resultado de la función `palabrasMatriz`, la matriz de los datos de las frecuencias referentes a las palabras clave que aparecen por cada elemento de `nombres`.
- `(autovalores, autovectores)`: el resultado de la función `eigDescomposicion`.
- `nAutoV`: el número de autovalores requeridos para que la varianza explicada por el análisis tenga al menos un porcentaje de valor `procentaje`.
- `pcaMatriz`: el resultado de `analisisComponentesPrincipales`, `matriz` proyectada sobre las nuevas componentes principales.
- `indNombre`: la posición de `nombre` en `nombres` en un `Int` en caso de que `nombre` se encuentre en la lista, si esto no es así, devolverá un mensaje de error.
- `sumaDeCuadrados`: una lista de valores `Double`, siendo estos la distancia euclídea cuadrada de las distintas observaciones con respecto a la observación dada `nombre`.
- `masCercanos`: es la respuesta de `indicesOrdenados` con respecto a `sumaDeCuadrados`. Recordemos que esta sería una lista de los índices de `sumaDeCuadrados` ordenados en función de lo pequeños que sean sus valores. Tendremos por lo tanto, los índices de `nombres` en función de lo cerca que estén de `nombre`.

Por último definimos el "*output*" como los primeros `n` nombres/títulos de la base de datos más cercanos, sin contar `nombre`. Usaremos `genericIndex` para que nos devuelva el elemento según su índice. El resultado que proporcionará `recomendador` es una lista `[String]` con los nombres o títulos de las `n` recomendaciones pedidas.

Código 4.22: Función `recomendador`

```
recomendador :: [(String,String)] -> [String] -> String -> Int -> Int ->
                                  Double -> [String]
recomendador datos palabrasVacias nombre n nClaves porcentaje =
   let claves = L.map fst (palabrasClave datos palabrasVacias nClaves)
       nombres = keys $ frecuencia $ L.map fst datos
       matriz = palabrasMatriz datos claves nombres
       (autovalores, autovectores) = eigDescomposicion matriz
       nAutoV = nAutovectores autovalores porcentaje
       pcaMatriz = analisisComponentesPrincipales autovectores matriz nAutoV
       indNombre = fromMaybe (error $ "Error: No se encuentra el elemento "
             ++ nombre ++ " en la base de datos.") (elemIndex nombre nombres)
       sumaDeCuadrados = LA.toList $ distEuclidCuadrPorFila pcaMatriz indNombre
       masCercanos = indicesOrdenados sumaDeCuadrados
   in L.map (\ind -> genericIndex nombres ind)
         (take n (L.filter (/= indNombre) masCercanos))
```

Código 4.22: Función `recomendador`

CAPÍTULO 5

Aplicaciones

Explicada la teoría del *clasificador Naive Bayes* y del *análisis de componentes principales*, y habiendo visto sus respectivas implementaciones en librerías en *Haskell*. Dedicaremos este capítulo a crear dos aplicaciones que servirán como ejemplos prácticos de sus utilidades.

Partiendo de una base de datos que dispondrá de fragmentos de textos, junto con el idioma en el que están escritos y un título que los identifique. La primera de las aplicaciones usará la *clasificación Naive Bayes* para determinar el idioma de una cadena de texto cualquiera dada. Mientras que la segunda, por medio del *análisis de componentes principales*, será capaz de dar una lista de títulos de la base de datos cuyo texto guarde mayor similitud con un fragmento de texto introducido.

Para ello será esencial generar una base de datos. Explicaremos el proceso de crear una base de datos que podamos usar en sendas aplicaciones. La base de datos deberá constar de texto en distintos idiomas y la capacidad de ser identificados cada uno con un nombre. El contenido del texto debe ser variado en cuanto a las palabras que lo forman y con una extensión relativamente considerable, para así asegurar el buen funcionamiento de nuestras aplicaciones. En este caso se trabajará con texto extraído de *Wikipedia*.

5.1– *Wikipedia*

Wikipedia se define en su entrada homónima en su web en español como *"una enciclopedia libre, políglota y editada de manera colaborativa"*. Conocida por casi todos los internautas, esta enciclopedia creada en 2001 se ha convertido en una de las mayores obras de internet, sustentada de manera independiente, siendo administrada por la *Fundación Wikimedia*, una organización sin ánimo de lucro financiada por donaciones, y con más de tres mil administradores y casi trescientos mil usuarios que ayudan y escriben para la actualización y la riqueza de una enciclopedia libre y global.

El origen del texto para alimentar la base de datos no ha sido una decisión fortuita. Debía de cumplir unos requisitos concretos:

- Variedad de idiomas.
- Posibilidad de descargar el texto, el idioma y un título representativo.

- Diversidad de palabras y temas.

Nos debíamos asegurar de que la fuente tuviera una variedad de idiomas considerable. En la mayoría del contenido en la red, el idioma predominante es el inglés; sin embargo *Wikipedia* cumple esto con creces, teniendo artículos en 334 idiomas y, lo que es más importante, no de manera anecdótica: tiene 30 idiomas con más de medio millón de artículos.

También era de vital importancia que la página web elegida, contara con una API (interfaz de programación de aplicaciones) que nos permitiera descargar el texto y un título para el mismo. Cosa que *Wikipedia* nos otorga. No hacía falta necesariamente un título, también se manejo la posibilidad de obtener un nombre de usuario por cada texto para relacionar la similitud entre usuarios, en definitiva, necesitábamos un pequeño grupo de palabras que nos hicieran identificar el texto.

Mucho menos usual en las páginas webs, es la posibilidad de saber el idioma en el que está el texto descargado, muchas no distinguen entre un idioma u otro, pero este no es el caso de *Wikipedia* que cuenta con un subdominio para cada idioma, de donde se puede extraer fácilmente el idioma. La diversidad de palabras también es algo en el que la *Wikipedia* sobresale, la enciclopedia libre cuenta con una gran cantidad y diversidad de temas en sus artículos.

El contenido del conjunto de entrenamiento del modelo vendrá de artículos de *Wikipedia*. Concretamente, utilizaremos introducciones de artículos elegidos de manera aleatoria, gracias a una herramienta que nos da *Wikipedia*. La decisión de usar solo la primera parte de los artículos viene motivada por las características de los artículos en sí. Muchos de ellos son artículos cortos y todo su contenido se encuentra en la introducción, además elegir sólo la introducción también nos simplifica el trabajo, sólo tiene texto simple, nos libramos de las posibles alteraciones en nuestra porción de palabras que pueden causar tablas, esquemas u otros formatos más allá del texto simple. Al fin y al cabo, sólo buscamos información de texto "real". Otro beneficio podría ser la mayor diversidad de origen de palabras, cogiendo solo introducciones nos libramos de posibles artículos más largos que puedan acumular una gran parte de las palabras de la base de datos, si los demás artículos son de menor tamaño.

Otra decisión que se tuvo que tomar es la elección de idiomas que nuestro clasificador pueda tratar. Aunque la idea más sencilla y lógica pudiera parecer a priori seleccionar los idiomas con más artículos, este número está distorsionado. Un ejemplo sorprendente de ello es que el segundo idioma con mayor número de artículos, muy cerca del inglés y con tres veces más artículos que el español, sea el cebuano, el sexagesimosegundo idioma más hablado del mundo con a penas 21 millones de hablantes y autóctono de la isla de Cebú en Filipinas. Esto parece ser que se debe a la existencia de *bots* que añaden artículos muy cortos de ciudades y pueblos de manera automática de este idioma y otros idiomas. Para ahorrarnos estos idiomas menos habituales, hemos decidido quedarnos con los idiomas con mayor número de usuarios activos como regla a la hora de seleccionar los más comunes en la web. Según *Wikipedia*, el idioma español a pesar de contar tres veces menos artículos que el cebuano, tiene 13.284 usuarios activos frente a los 172 del cebuano.

Todos estos datos relacionados con el número de usuarios o artículos de *Wikipedia* están a nuestra disposición en una entrada de la versión inglesa de la propia *Wikipedia*, llamada "*List of Wikipedias*". Fuente debidamente citada en la bibliografía.

5.2– Preparación de los datos

El paso previo a la creación de las aplicaciones es la constitución de una base de datos, en este caso, una base de datos con forma de tabla donde se guarden las introducciones de distintos artículos de *Wikipedia* junto con su título y el idioma correspondiente en el que están escritos.

5.2.1. Generación aleatoria de artículos de *Wikipedia*

La selección de los artículos que guardaremos en nuestra base de datos se hará de manera totalmente aleatoria. En esta subsección diseñamos este proceso aleatorio.

Toda esta primera implementación estará ubicada en un archivo que hemos llamado `GeneradorEnlaces.hs`. Lo primero que debemos hacer en este archivo será declarar el módulo que estamos creando, `GeneradorEnlaces`, marcando en él las funciones `enlacesAleatoriosPorIdioma` y `enlacesAleatorios` que querremos exportar para usar en otros programas:

Código 5.1: Declaración del módulo `GeneradorEnlaces`

```
1 module GeneradorEnlaces (
2     enlacesAleatoriosPorIdioma,
3     enlacesAleatorios
4     )where
```

Código 5.1: Declaración del módulo `GeneradorEnlaces`

Será fundamental importar las siguientes librerías:

- Para evitar ambigüedades, ocultaremos la función estándar `id` del módulo `Prelude`, importado por defecto.
- `Network.HTTP.Client` y `Network.HTTP.Client.TLS` para hacer las peticiones correspondientes a la API.
- `Data.Aeson` donde encontraremos herramientas para tratar los datos en formato JSON y convertirlos a tipos de datos más manejables.
- `GHC.Generics` nos aportará la derivación automática de instancias.

Código 5.2: Librerías usadas en `GeneradorEnlaces`

```
1 import Prelude hiding (id)
2 import Network.HTTP.Client
3 import Network.HTTP.Client.TLS
4 import Data.Aeson
5 import GHC.Generics
```

Código 5.2: Librerías usadas en `GeneradorEnlaces`

Los tipos de datos están determinados por el formato que nos proporciona la API. Tendremos 3 tipos, cada uno dentro del siguiente, `RespuestaApi` es el nos devolverá la API con un sólo campo llamado `query` de tipo `BusquedaAleatoria` que nos indicará con su único campo el tipo de busqueda que hemos hecho en la API, “random”. Este campo

`random` es una lista del último tipo de dato, `ArticuloAleatorio`, cuyo único campo `id` nos aportará un número de tipo `Int` que usaremos para identificar el artículo elegido de manera aleatoria.

Código 5.3: Tipos de datos usados

```
data ArticuloAleatorio = ArticuloAleatorio
    { id   :: Int
    } deriving (Show, Generic)

data BusquedaAleatoria = BusquedaAleatoria
    { random :: [ArticuloAleatorio]
    } deriving (Show, Generic)

data RespuestaApi = RespuestaApi
    { query :: BusquedaAleatoria
    } deriving (Show, Generic)
```

Código 5.3: Tipos de datos usados

Las siguientes 3 líneas de código van destinadas a la deserialización de datos en formato JSON, para convertirlos en nuestros datos ya definidos.

Código 5.4: Instancias usadas

```
instance FromJSON ArticuloAleatorio
instance FromJSON BusquedaAleatoria
instance FromJSON RespuestaApi
```

Código 5.4: Instancias usadas

La API que proporciona *Wikipedia* da la posibilidad de obtener de manera aleatoria unos identificadores (`id`) que podremos usar para encontrar estos artículos en el próximo módulo. Sin embargo, la elección del idioma no es aleatoria y debe hacerse desde el subdominio de la url de la llamada de la API. Esto nos ofrece algunas ventajas como que podremos asegurarnos de que tenemos en nuestra base de datos texto proveniente del mismo número de artículos. Si la elección del idioma fuera aleatoria, tendríamos una gran cantidad de texto de los idiomas que más artículos tengan en *Wikipedia* y nuestro clasificador podría fallar en algunos idiomas.

La parte negativa es que no podremos estimar la frecuencia con la que los artículos tienen un idioma u otro a partir de los datos obtenidos, ya que todos los idiomas tendrán el mismo número de artículos. Esto podría llegar a ser un problema en otros contextos pero en este caso, consultando la entrada ya mencionada *List of Wikipedias*, tenemos a nuestro alcance datos sobre el número de usuarios activos por idioma. Posteriormente, en el desarrollo de la aplicación usaremos estos datos para determinar los idiomas que usaremos y la frecuencia de cada uno de ellos.

Llegamos a nuestra primera función, `enlacesAleatoriosPorIdioma` hará un llamamiento a la API de *Wikipedia* para elegir de manera aleatoria un número concreto de artículos, `nArticulos`, en un idioma concreto que vendrá indicado por `idioma`. `idioma` será un `String` formado por un subdominio de *Wikipedia* identificado con un idioma (`"en"`, `"es"`, `"fr"`, etc). Los códigos de los idiomas usados forman parte de la lista *ISO 639-1*, la lista de abreviaturas de dos letras de idiomas más usada globalmente y la que utiliza *Wikipedia* en sus subdominios.

La función devolverá una lista con una dupla con dos `String`, con el código del idioma y el identificador (`id`) del artículo seleccionado de manera aleatoria. Cada código hará referencia a un artículo concreto, del que posteriormente descargaremos el texto haciendo otra llamada a la API.

Código 5.5: Función `enlacesAleatoriosPorIdioma`

```
enlacesAleatoriosPorIdioma :: Int -> String -> IO [(String,String)]
enlacesAleatoriosPorIdioma nArticulos idioma = do
    manager <- newManager tlsManagerSettings
    let url = "https://" ++ idioma ++ ".wikipedia.org/w/api.php?action=query"++
              "&format=json&list=random&rnnamespace=0&rnlimit="++(show nArticulos)
    req <- parseRequest url
    res <- httpLbs req manager
    let maybeResponse = decode (responseBody res) :: Maybe RespuestaApi
    return $ case maybeResponse of
        Just respuestaApi ->
            fmap (\page -> (idioma,show (id page))) (random (query respuestaApi))
        Nothing -> []
```

Código 5.5: Función `enlacesAleatoriosPorIdioma`

Terminamos este módulo con una simple función `enlacesAleatorios` que nos devuelva el resultado de `enlacesAleatoriosPorIdioma` aplicada a todos los idiomas de la lista `idiomas`. Elegiremos aleatoriamente un número `nArticulos` de artículos por cada idioma, el número de artículos por llamada será limitado por quinientos, si `nArticulos` supera este límite se realizará una recursión por la que haremos llamamientos a la función `enlacesAleatoriosPorIdioma`, y por ende a la API, con un máximo de quinientos artículos hasta llegar al número `nArticulos` requerido. Gracias a ser `Haskell` un lenguaje funcional, podremos usar la función `enlacesAleatorios` como si fuera una lista.

Código 5.6: Función `enlacesAleatorios`

```
enlacesAleatorios :: Int -> [String] -> IO [(String,String)]
enlacesAleatorios nArticulos idiomas =
   if nArticulos > 500
   then do resto <- enlacesAleatorios (nArticulos-500) idiomas
           resultadoAux <- mapM (\x -> enlacesAleatoriosPorIdioma 500 x) idiomas
           return ((concat resultadoAux) ++ resto)
   else do resul <- mapM (\x -> enlacesAleatoriosPorIdioma nArticulos x) idiomas
           return (concat resul)
```

Código 5.6: Función `enlacesAleatorios`

5.2.2. Extracción del texto de los artículos

El módulo `GeneradorTextos` tendrá como objetivo convertir la lista de duplas de códigos de idioma e identificadores de artículos en otra lista de tuplas de tres elementos con el mismo código de idioma, el título del artículo y la introducción del mismo.

El nuevo archivo `GeneradorTextos.hs` comenzará con la declaración del módulo, indicando futuras exportaciones.

Código 5.7: Declaración del módulo `GeneradorTextos`

```
module GeneradorTextos (
    Wiki,
    idioma,
    titulo,
    texto,
    textosAleatorios,
    extractorIntroTitulo
    )where
```

Código 5.7: Declaración del módulo `GeneradorTextos`

Necesitaremos alguna librería más en esta ocasión, concretamente:

- Importaremos el módulo que hicimos para disponer de `enlacesAleatorios`.
- También necesitaremos todas las librerías del anterior "script" para volver a hacer llamamientos a la API y tratar los resultados. Estas serían `Network.HTTP.Client`, `Network.HTTP.Client.TLS`, `Data.Aeson` y `GHC.Generics`.
- `Data.HashMap.Strict` será útil para manejar los datos con estructura de HashMap, que será como se descargarán algunos datos de la API. La importación la haremos con `import qualified Data.HashMap.Strict as HM` para ahorrarnos problemas de ambigüedades y poder acceder a los elementos de la librería de manera más simple con el prefijo `HM`.
- La estructura de datos `Text` la manipularemos con `Data.Text`, librería que marcaremos con `T`.
- Ajustaremos el texto descargado con herramientas de `Data.Char` para quedarnos sólo con palabras y eliminar símbolos u otros caracteres del texto plano que no sean palabras.

Código 5.8: Librerías usadas en `GeneradorTextos`

```
import GeneradorEnlaces
import Network.HTTP.Client
import Network.HTTP.Client.TLS
import Data.Aeson
import GHC.Generics
import qualified Data.HashMap.Strict as HM
import qualified Data.Text as T
import Data.Char
```

Código 5.8: Librerías usadas en `GeneradorTextos`

Esta vez tendremos que la API nos proporcionará los datos con estructura `RespuestaApi`, un tipo de dato con un único campo llamado `query` de tipo `Busqueda` que a su vez tendrá otro campo de `pages` con estructura de `HashMap`. Los elementos de tipo `HashMap` almacenan datos en pares de clave-valor, en nuestro caso una clave de tipo `T.Text` y un valor de tipo `Articulo` que definiremos con un campo `extract` de tipo `T.Text` donde se descargará el texto plano de la introducción del artículo, y un campo `title` donde hará lo propio el

título. El operador `!` se usa para marcar que ese campo debe ser evaluado estrictamente, no perezosamente, para ahorrarnos posibles problemas de memoria.

Por último tenemos un tipo que no pertenece al de los datos descargados de la API sino que lo hemos creado nosotros para almacenar los datos que saquemos y conseguir así una semántica más clara. Hemos definido una estructura de datos `Wiki` con tres campos `idioma`, donde irá el código del idioma, `titulo` con el título del artículo y `texto` en referencia al texto que sacaremos de cada artículo.

Código 5.9: Tipos de datos usados

```
data RespuestaApi = RespuestaApi
    { query :: Busqueda
    } deriving (Show, Generic)

data Busqueda = Busqueda
    { pages :: HM.HashMap T.Text Articulo
    } deriving (Show, Generic)

data Articulo = Articulo
    { title :: !T.Text,
      extract :: !T.Text
    } deriving (Show, Generic)

data Wiki = Wiki
    { idioma :: !T.Text,
      titulo :: !T.Text,
      texto :: !T.Text
    } deriving (Show,Generic)
```

Código 5.9: Tipos de datos usados

Así quedarían las instancias:

Código 5.10: Instancias usadas

```
instance FromJSON RespuestaApi
instance FromJSON Busqueda
instance FromJSON Articulo
```

Código 5.10: Instancias usadas

La función principal del módulo es `extractorIntroIdiomas`, a partir de un (`String`, `String`) donde el primer `String` representa un código de idioma y el segundo un identificador de un artículo de *Wikipedia*, y tras hacer un llamamiento a la API, recoger los datos en `res`, y decodificarlos en `maybeRespuestaApi` en un formato `Maybe RespuestaApi`, conseguimos un resultado en formato `Either String Wiki` donde la salida será un `String` indicando de un error si el código era erróneo, o simplemente `maybeRespuestaApi` está vacío por otro tipo de error, o un elemento `Wiki` que guarda el idioma, el título y el texto ya manipulado con `limpiadorIntroduccion`, una función que definiremos más adelante, que nos dejará el texto de la introducción del artículo "limpio".

Código 5.11: Función `extractorIntroIdiomas`

```
extractorIntro :: (String,String) -> IO (Either String Wiki)
extractorIntro (idioma, articuloID) = do
```

```
    manager <- newManager tlsManagerSettings
    let url = "https://" ++ idioma ++
              ".wikipedia.org/w/api.php?action=query&prop=info|extracts"
            ++ "&explaintext&exintro&format=json&pageids=" ++ articuloID
    req <- parseRequest url
    res <- httpLbs req manager
    let maybeRespuestaApi = decode (responseBody res) :: Maybe RespuestaApi
    return $ case maybeRespuestaApi of
        Just respuestaApi ->
            let articulo = Prelude.head $ HM.elems $ pages $ query respuestaApi
                texto = extract articulo
                textoLimpio = limpiadorIntroduccion texto
                titulo = title articulo
            in Right (Wiki (T.pack idioma) titulo textoLimpio)
        Nothing -> Left $ ("Texto no encontrado")
```

Código 5.11: Función `extractorIntroIdiomas`

Además de `extractorIntroIdiomas` que conseguirá descargar toda la información necesaria y guardarla en un dato con estructura `Wiki` por medio del idioma y el código del artículo, definimos una versión alternativa que hará lo mismo pero a través de un `[(String,String)]` con el dato del idioma y el título directamente. La única diferencia será la llamada a la API que utilizará el título en vez del código para identificar los artículos.

Esta función será utilizada en la aplicación del *análisis de componentes principales*, donde buscaremos similitudes entre los artículos recogidos de manera aleatoria y uno del que el usuario dará el título. La función se usará para obtener la introducción de este artículo no aleatorio.

Código 5.12: Función `extractorIntroTitulo`

```
extractorIntroTitulo :: (String,String) -> IO (Either String Wiki)
extractorIntroTitulo (idioma, titulo) = do
    manager <- newManager tlsManagerSettings
    let url = "https://" ++ idioma ++
              ".wikipedia.org/w/api.php?action=query&prop=info|extracts"
            ++ "&explaintext&exintro&format=json&titles=" ++ titulo
    req <- parseRequest url
    res <- httpLbs req manager
    let maybeRespuestaApi = decode (responseBody res) :: Maybe RespuestaApi
    return $ case maybeRespuestaApi of
        Just respuestaApi ->
            let articulo = Prelude.head $ HM.elems $ pages $ query respuestaApi
                texto = extract articulo
                textoLimpio = limpiadorIntroduccion texto
                tituloAux = title articulo
            in Right (Wiki (T.pack idioma) tituloAux textoLimpio)
        Nothing -> Left $ ("Texto no encontrado")
```

Código 5.12: Función `extractorIntroTitulo`

Definiremos a continuación la función `limpiadorIntroduccion` que "limpiará" el texto de las introducciones. Sabiendo que la introducción adquirida en formato de texto plano

tendrá `"\n"` indicando el final de cada párrafo, cambiaremos esto por un espacio en blanco para poder separar la última palabra de cada párrafo con la primera del siguiente. Además, estrenamos la librería `Data.Char` para filtrar el texto, quitar todo lo que no sean letras o espacios en blanco y pasar todas las letras a minúsculas.

Código 5.13: Función `limpiadorIntroduccion`

```
limpiadorIntroduccion :: T.Text -> T.Text
limpiadorIntroduccion text =
    let introLimpia = T.replace (T.pack "\n") (T.pack " ") text
    in T.pack (Prelude.map toLower (Prelude.filter (\c -> isAlpha c || c == ' ')
                                                   (T.unpack ( introLimpia))))
```

Código 5.13: Función `limpiadorIntroduccion`

Terminamos con una función similar a la última del anterior módulo, extiende la función principal `extractorIntroIdiomas` a todos los elementos de `enlacesAleatorios`, importada desde el otro módulo.

En `textosAleatorios` tendremos una lista de datos con estructura `Wiki` que almacenarán idioma, título e introducción de unos artículos de *Wikipedia* elegidos de manera aleatoria.

Código 5.14: Función `textosAleatorios`

```
textosAleatorios :: Int -> [String] -> IO [Either String Wiki]
textosAleatorios nArticulos idiomas = do
   aux <- enlacesAleatorios nArticulos idiomas
   mapM extractorIntro aux
```

Código 5.14: Función `textosAleatorios`

5.2.3. Creación de nuestra base de datos

`GeneradorBaseDatos`, ubicado en un nuevo archivo homónimo, será nuestro último módulo dedicado a la construcción de la base de datos. Desarrollaremos en él una serie de funciones que usaremos para crear dos bases de datos para cada una de las dos aplicaciones.

Usaremos una base de datos SQLite. SQlite es un tipo de base de datos que no requiere de una conexión con un servidor para almacenar los datos, lo hace en un solo archivo del sistema de archivos local.

Comenzamos el módulo con la declaración como siempre.

Código 5.15: Declaración del módulo `GeneradorBaseDatos`

```
module GeneradorBaseDatos where
```

Código 5.15: Declaración del módulo `GeneradorBaseDatos`

Importamos el anterior módulo `GeneradorTextos`. Además de 2 nuevas librerías sobre las base de datos SQLite, `Database.HDBC` y `Database.HDBC.Sqlite3`, y otras 2 para obtener herramientas que necesitaremos en el desarollo del módulo, la primera sobre las listas, `Data.List`, y otra sobre los tipos de datos `Either`, `Data.Either`.

Código 5.16: Librerías usadas en `GeneradorBaseDatos`

```
import GeneradorTextos
import Database.HDBC
import Database.HDBC.Sqlite3
import Data.List
import Data.Either
```

Código 5.16: Librerías usadas en `GeneradorBaseDatos`

Nuestra primera función, `crearBaseDatos` creará una base de datos de un nombre dado como `String` en `archivo`. En esta base de datos instauramos una tabla de un nombre también dado como `String`, `nombreTabla`. La tabla tendrá tres columnas donde introduciremos nuestros datos, estas columnas tendrán datos con estructura `Text` y las llamaremos `idioma`, `titulo` y `texto`. La función imprimirá un mensaje dejando ver que funcionó correctamente.

Código 5.17: Función `crearBaseDatos`

```
crearBaseDatos :: String -> String -> IO()
crearBaseDatos archivo nombreTabla = do
  conn <- connectSqlite3 archivo
  run conn createStatement []
  commit conn
  disconnect conn
  putStrLn "Base de datos creada con éxito."
  where createStatement = "CREATE TABLE " ++ nombreTabla ++
                          " (idioma TEXT, titulo TEXT, texto TEXT)"
```

Código 5.17: Función `crearBaseDatos`

Creada la base de datos, la alimentaremos con datos de tipo `Wiki`. La función `insertarTextoEnBaseDatos` inserta en la tabla de nombre `nombreTabla` de la base de datos de nombre `archivo` un dato con estructura `Wiki`, donde cada uno de ellos irá a su columna correspondiente.

Código 5.18: Función `insertarTextoEnBaseDatos`

```
insertarTextoEnBaseDatos :: Wiki -> String -> String -> IO()
insertarTextoEnBaseDatos wiki archivo nombreTabla = do
  conn <- connectSqlite3 archivo
  stmt <- prepare conn insertStatement
  execute stmt [toSql (idioma wiki), toSql (titulo wiki), toSql (texto wiki)]
  commit conn
  disconnect conn
  where insertStatement = "INSERT INTO " ++ nombreTabla ++ " VALUES (?, ?, ?)"
```

Código 5.18: Función `insertarTextoEnBaseDatos`

La siguiente función será otra versión de la anterior enfocada a meter en la base de datos un cierto artículo del que tenemos el nombre. Esta función, como ya se mencionó, tendrá su utilidad en la aplicación del *análisis de componentes principales*. La diferencia que tiene con la anterior función es el uso de la función del anterior módulo `extractorIntroTitulo`, para adquirir el dato `Wiki` del artículo de nombre e idioma definidos en `articulo`, de estructura `(String,String)` siendo el primer elemento referente al idioma y el segundo al título. Estos datos serán introducidos en una tabla `nombreTabla` de una base de datos `archivo`.

Código 5.19: Función `insertarTextoEnBaseDatosArticulo`

```
insertarTextoEnBaseDatosArticulo :: (String, String) -> String -> String -> IO()
insertarTextoEnBaseDatosArticulo articulo archivo nombreTabla = do
  conn <- connectSqlite3 archivo
  stmt <- prepare conn insertStatement
  maybeWiki <- extractorIntroTitulo articulo
  case maybeWiki of
      Left error -> putStrLn "Texto no encontrado"
      Right wiki -> do
        execute stmt [toSql (idioma wiki), toSql (titulo wiki),
                                        toSql (texto wiki)]
        commit conn
        disconnect conn
        putStrLn "Insertado en la base de datos con éxito."
  where insertStatement = "INSERT INTO " ++ nombreTabla ++ " VALUES (?, ?, ?)"
```

Código 5.19: Función `insertarTextoEnBaseDatosArticulo`

La última función, `baseDatosAleatoria`, recopila las funciones `crearBaseDatos` para crear una base de datos con una tabla de nombres dados, e `insertarTextoEnBaseDatos` donde se introducirá en la base de datos creada los `Wikis` escogidos de manera aleatoria por `textosAleatorios`.

La función `baseDatosAleatoria`, además de requerir los nombres de la base de datos y su tabla, necesitará de un número de artículos, `nArticulos`, y una lista de idiomas, `idiomas`. El resultado será un archivo de nombre `archivo` que tendrá almacenados `nArticulos` artículos en todos los idiomas de la lista `idiomas`. También se indicará el número de artículos introducidos en la base de datos.

Código 5.20: Función `baseDatosAleatoria`

```
baseDatosAleatoria :: String -> String -> Int -> [String] -> IO()
baseDatosAleatoria archivo nombreTabla nArticulos idiomas = do
  textos <- textosAleatorios nArticulos idiomas
  crearBaseDatos archivo nombreTabla
  mapM_ (\texto -> either putStrLn
        (\x -> insertarTextoEnBaseDatos x archivo nombreTabla) texto) textos
  let exitos = length (rights textos)
  putStrLn $ "Se han insertado " ++ show exitos ++
            " elementos en la base de datos con éxito."
```

Código 5.20: Función `baseDatosAleatoria`

Sacaremos de este módulo dos funciones fundamentales para las aplicaciones que presentaremos a continuación.

- `baseDatosAleatoria` que creará una base de datos y la rellenará de un número elegido de introducciones con su título e idioma de tantos idiomas como se indique. Siendo todas las introducciones recopiladas de manera aleatoria.

- `insertarTextoEnBaseDatosArticulo` que añadirá a una tabla de una base de datos ya creada los datos de un artículo de idioma y título dado.

Herramientas auxiliares para acceder a la base de datos

Antes de entrar en la propia construcción de las aplicaciones, definiremos dos funciones auxiliares que nos ayudarán a manipular los datos provenientes de la base de datos. Las definiremos en el módulo `GeneradorBaseDatos` que importaremos en los ficheros de ambas, para no tener que definirlas directamente en ellos.

La primera es `extraerColumna` que devuelve una columna concreta de una tabla SQL:

Código 5.21: Función `extraerColumna`

```
1 extraerColumna :: [[SqlValue]] -> Integer -> [String]
2 extraerColumna sql num =
3   map (\col -> fromSql $ genericIndex col num :: String) sql
```

Código 5.21: Función `extraerColumna`

Para interactuar con una base de datos SQLite tenemos `buscadorBaseDatos`. La función nos devolverá los valores de tipo SQL correspondientes a la petición que se haga en `busqueda`.

Código 5.22: Función `buscadorBaseDatos`

```
1 buscadorBaseDatos archivo busqueda = do
2   conn <- connectSqlite3 archivo
3   res <- quickQuery' conn busqueda []
4   disconnect conn
5   return res
```

Código 5.22: Función `buscadorBaseDatos`

5.3– Aplicación: Clasificador idioma

En la anterior sección creamos las herramientas necesarias para el desarrollo de una base de datos que tenga almacenada introducciones de artículos de *Wikipedia* elegidos de manera aleatoria con su idioma y título. Aunque para esta aplicación, sólo necesitaremos las columnas `idioma` y `texto`. Para el clasificador de idiomas basado en la *clasificación de Naive Bayes*, necesitaremos tener a nuestra disposición una base de datos que almacene introducciones con su idioma, de las que extraeremos las palabras de cada idioma.

En esta nueva sección, daremos los últimos retoques a nuestra base de datos e implementaremos una aplicación capaz de clasificar un fragmento de texto cualquiera según el idioma en el que se encuentra.

Una vez tengamos los datos organizados, sólo quedaría pasarlos al formato adecuado para poder usarlos en las funciones desarrolladas en la librería `Clasificador.hs`. La aplicación final la implementaremos en un nuevo archivo llamado `ClasificadorIdioma.hs`.

5.3.1. Declaración del módulo e importación de librerías

Creamos y declaramos un nuevo módulo llamado `ClasificadorIdioma`, sobre él implementaremos el clasificador:

Código 5.23: Declaración del módulo `ClasificadorIdioma`

```
1 module ClasificadorIdioma where
```

Código 5.23: Declaración del módulo `ClasificadorIdioma`

Algunas de las librerías que necesitaremos son ya conocidas, por lo que no nos excederemos en su presentación:

- `GeneradorBaseDatos` es la librería creada en la anterior sección que usaremos para extraer los datos necesarios.
- `Clasificador` es la librería en la que hemos implementado todas las herramientas relacionadas con la *clasificación de Naive Bayes*.
- `Data.List` nos aportará funciones para manejar las listas, usamos en su importación un alias, `L`, para confrontar posibles errores de ambigüedad con otras funciones de mismo nombre de otras librerías.
- `Data.Hashable` nos ayuda a crear datos con estructura `HashMap`, implementados desde la librería `Data.HashMap.Strict`, ya utilizada en la creación de la base de datos.
- `Data.Ord` nos proporciona funcionalidades relacionadas con el orden de los elementos que nos serán útiles.
- `Data.HDBC.Sqlite3` y `Data.HDBC` serán esenciales para usar el controlador HDBC para acceder y consultar a bases de datos SQLite3 como `textosIdiomas.sql`.
- `Data.Char` la usaremos para modificar datos de texto tipo `String`.

Código 5.24: Librerías usadas en `ClasificadorIdioma`

```
1 import GeneradorBaseDatos
2 import Clasificador
3 import Data.List as L
4 import Data.Hashable
5 import Data.HashMap.Strict as HM
6 import Data.Ord
7 import Database.HDBC.Sqlite3
8 import Database.HDBC
9 import Data.Char
```

Código 5.24: Librerías usadas en `ClasificadorIdioma`

5.3.2. Elección de idiomas

Como ya se adelanto, tendremos que importar los idiomas que usaremos, más concretamente sus códigos *ISO 639-1*, y los datos de frecuencia de uso de cada uno de ellos de manera manual.

Hemos decidido usar los 30 idiomas con mayor cantidad de usuarios activos en *Wikipedia*, la elección de la cifra de 30 idiomas es totalmente arbitraria, se recomienda que se usen idiomas comunes para que el texto extraído tenga mayor riqueza y tamaño.

Hemos almacenado los códigos de estos 30 idiomas y el número de usuarios activos por idioma en `frecUsuariosPorIdioma` en formato `HashMap` donde la clave es el código de cada idioma y el valor es el número de usuarios.

Idioma	Código	Idioma	Código
Inglés	en	Turco	tr
Alemán	de	Indonesio	id
Francés	fr	Checo	cs
Español	es	Coreano	ko
Japonés	ja	Sueco	sv
Ruso	ru	Vietnamita	vi
Portugués	pt	Finés	fi
Italiano	it	Húngaro	hu
Chino	zh	Tailandés	th
Persa	fa	Bengalí	bn
Polaco	pl	Noruego	no
Árabe	ar	Catalán	ca
Neerlandés	nl	Hindi	hi
Hebreo	he	Griego	el
Ucraniano	uk	Rumano	ro

Tabla 5.1: Códigos de idiomas ISO 639-1

Idioma	Nº usuarios	% total	Idioma	Nº usuarios	% total
Inglés	115.659	45,98 %	Turco	2.581	1,03 %
Alemán	16.787	6,67 %	Indonesio	2.567	1,02 %
Francés	15.990	6,36 %	Checo	1.994	0,79 %
Español	13.430	5,34 %	Coreano	1.978	0,79 %
Japonés	12.651	5,03 %	Sueco	1.955	0,78 %
Ruso	8.752	3,48 %	Vietnamita	1.575	0,63 %
Portugués	8.256	3,28 %	Finés	1.535	0,61 %
Italiano	7.203	2,86 %	Húngaro	1.407	0,56 %
Chino	7.153	2,84 %	Tailandés	1.341	0,53 %
Persa	5.071	2,02 %	Bengalí	1.335	0,53 %
Polaco	4.253	1,69 %	Noruego	1.079	0,43 %
Árabe	3.766	1,50 %	Catalán	967	0,38 %
Neerlandés	3.425	1,36 %	Hindi	959	0,38 %
Hebreo	3.276	1,30 %	Griego	862	0,34 %
Ucraniano	2.975	1,18 %	Rumano	784	0,31 %

Tabla 5.2: Idiomas con mayor número de usuarios activos en *Wikipedia*

Código 5.25: `HashMap frecUsuariosPorIdioma`

```
1 frecUsuariosPorIdioma = HM.fromList [("en",115659),("de",16787),("fr",15990),
2   ("es",13430),("ja",12651),("ru",8752),("pt",8256),("it",7203),("zh",7153),
3   ("fa",5071),("pl",4253),("ar",3766),("nl",3425),("he",3276),("uk",2975),
4   ("tr",2581),("id",2567),("cs",1994),("ko",1978),("sv",1955),("vi",1575),
5   ("fi",1535),("hu",1407),("th",1341),("bn",1335),("no",1079),("ca",967),
6   ("hi",959),("el",862),("ro",784)]
```

Código 5.25: `HashMap frecUsuariosPorIdioma`

5.3.3. Generación base de datos

Con los idiomas que manejará el clasificador ya definidos, usaremos las herramientas diseñadas en la librería `GeneradorBaseDatos` para generar una base de datos para este problema concreto. Para esta tarea utilizaremos una función que definiremos como `introsIdiomas`, que generará una base de datos con un número dado `nIntros` de introducciones de artículos de *Wikipedia* en los 30 idiomas seleccionados anteriormente.

Hemos dejado la lista de los idiomas implementada de manera directa para intentar facilitar el uso de la función, sobre todo cuando llegue el momento de usar los datos de frecuencia. Sin embargo, el número de introducciones sí será un valor que el usuario introducirá para fabricar los datos. Es importante tener en mente que el número de introducciones total será `nIntros` multiplicado por 30.

La función `introsIdiomas` ejecutará `baseDatosAleatoria` de la librería `GeneradorBaseDatos`, creando así un archivo llamado `introsIdiomas.sql` donde tendremos una tabla que llamaremos `tablaIdiomas` con un número `nIntros` de introducciones de cada idioma en `frecUsuariosPorIdioma`.

Código 5.26: Función `introsIdiomas`

```
1 introsIdiomas nIntros = do
2   baseDatosAleatoria "introsIdiomas.sql" "tablaIdiomas" nIntros idiomas
3   where idiomas = HM.keys frecUsuariosPorIdioma
```

Código 5.26: Función `introsIdiomas`

Lista `intros`

Accederemos a la base de datos `introsIdiomas.sql` con la función `intros`, para ello usamos las 2 funciones definidas en la subsección anterior. Después de obtener una lista creada a partir de columnas de la tabla de la base de datos substraída con ayuda de `buscadorBaseDatos` y la orden correcta. Usamos la función `frecuencia`, de la librería `clasificador.hs`, para quedarnos con un `HasMap` con cada introducción y su idioma como código, y las veces que se repiten en la base de datos como valor. Por lo que si aplicamos la herramienta `keys` de `Data.HashMap.Strict`, obtendremos sólo los códigos, es decir, una lista de elementos de estructura `(String,String)` con el idioma y una introducción, sin repeticiones ya que los códigos no se pueden repetir.

Código 5.27: Función `intros`

```
1 intros = do
2   baseDatos <- buscadorBaseDatos
3             "introsIdiomas.sql" "SELECT idioma, texto FROM tablaIdiomas"
4   let introsAux = zip (extraerColumna baseDatos 0) (extraerColumna baseDatos 1)
5   return $ HM.keys $ frecuencia introsAux
```

Código 5.27: Función `intros`

5.3.4. Clasificadores de idiomas

Con todos los preparativos terminados, se pueden definir las funciones definitivas que ejercerán de clasificadores de idiomas. Tendremos 2 versiones.

Primero presentaremos `clasificadorIdioma`, que a partir de un fragmento de texto en formato `String`, nos devolverá otro `String` con el código *ISO 639-1* que el programa le asignó.

Para ello, la función guardará `intros` en `datos`. Usaremos esa información para definir el resto de parámetros que necesitamos para `clasificadorClases`, importado desde la librería que diseñamos previamente. El único dato que no calcularemos será la frecuencia que la teníamos almacenada de manera explícita en `frecUsuariosPorIdioma`. Recibiremos una lista con todos los idiomas y su verosimilitud, elegimos la más alta y terminamos mostrando el código que representa el idioma que nuestro clasificador considera que debe ser asignado.

Código 5.28: Función `clasificadorIdioma`

```
clasificadorIdioma :: String -> IO String
clasificadorIdioma texto = do
    datos <- intros
    let clases = HM.keys $ (frecuencia . L.map fst) datos
        nPalabras = fromInteger $ sum $ HM.elems
                  ((frecuencia . concatMap words) (L.map snd datos))
        nExtractos = sum $ HM.elems frecUsuariosPorIdioma
        probs = clasificadorClases texto datos frecUsuariosPorIdioma
                                          clases nPalabras nExtractos
    return $ fst (L.maximumBy (comparing snd) probs)
```

Código 5.28: Función `clasificadorIdioma`

La segunda versión es más completa, sólo variará en la última línea, donde en vez de quedarse con el idioma con mayor verosimilitud, devolverá una lista en formato `[(String, Double)]`, donde el primer valor del par es el código del idioma y el segundo la respectiva verosimilitud. El orden de la lista no será arbitrario, irá ordenada de mayor verosimilitud a menor.

Código 5.29: Función `clasificadorIdiomaTodos`

```
clasificadorIdiomaTodos texto = do
   datos <- intros
   let clases = HM.keys $ (frecuencia . L.map fst) datos
       nPalabras = fromInteger $ sum $ HM.elems
                 ((frecuencia . concatMap words) (L.map snd datos))
       nExtractos = sum $ HM.elems frecUsuariosPorIdioma
       probs = clasificadorClases (L.map toLower texto) datos
                       frecUsuariosPorIdioma clases nPalabras nExtractos
   return (L.sortBy (comparing (Down . snd)) probs)
```

Código 5.29: Función `clasificadorIdiomaTodos`

5.4– Aplicación: Recomendador artículos

El desarrollo del **recomendador de artículos** que montaremos guarda similitudes con la anterior aplicación. En esta ocasión, crearemos una base de datos con un mayor número de introducciones pero de un sólo idioma, nos interesará comparar los artículos entre ellos y para ello deben estar en el mismo idioma. Por lo tanto, sólo nos interesaran las columnas de la tabla de la base de datos relativas al título y el texto.

La misión de la aplicación que escribiremos en el fichero `RecomendadorTitulos.hs`, será dar un uso práctico a las funciones generales que definimos en `Recomendador.hs`. Utilizaremos el clasificador que diseñamos con las bases teóricas del *análisis de componentes principales* para crear un recomendador que, a partir de cualquier título de un artículo web de *Wikipedia* que proporcione el usuario, devuelva una lista de los artículos de la base de datos que tienen mayor similitud con el dado. Teniendo la base de datos sus artículos recogidos de manera aleatoria.

5.4.1. Declaración del módulo e importación de librerías

Crearemos un módulo con el mismo nombre que dimos al archivo.

Código 5.30: Declaración del módulo `RecomendadorArticulos`

```
1 module RecomendadorArticulos where
```

Código 5.30: Declaración del módulo `RecomendadorArticulos`

Entre las librerías importadas, destacaremos las dos creadas para este trabajo, `GeneradorBaseDatos` y `Recomendador`. Además tendremos `Data.List` y `Data.HashMap.Strict` para operar sobre listas y datos `HashMap` respectivamente, y `Database.HDBC.Sqlite3` y `Database.HDBC` que usaremos para hacer consultas a la base de datos.

Código 5.31: Librerías usadas en `RecomendadorArticulos`

```
1 import GeneradorBaseDatos
2 import Recomendador
3 import Data.List as L
4 import Data.HashMap.Strict as HM
5 import Database.HDBC.Sqlite3
6 import Database.HDBC
```

Código 5.31: Librerías usadas en `RecomendadorArticulos`

5.4.2. Generación base de datos

Las dos siguientes funciones estarán dedicadas a crear la base de datos. La función `introsArticulos` creará una base de datos de tantos artículos como se especifique en `nIntros` y el idioma introducido por su código *ISO 639-1* en `idioma`. Usará la función `baseDatosAleatoria` de `GeneradorBaseDatos`. El nombre que le hemos asignado a la base de datos SQL es `introsArticulos.sql` donde se creará una tabla `tablaArticulos`.

Código 5.32: Función `introsArticulos`

```
1 introsArticulos :: Int -> String -> IO ()
2 introsArticulos nIntros idioma = do
```

```
3    baseDatosAleatoria "introsArticulos.sql" "tablaArticulos" nIntros [idioma]
```

Código 5.32: Función `introsArticulos`

Se usará `insertarArticulo` para introducir en la tabla de la base de datos creada, `introsArticulos` un artículo cualquiera. La información que daremos de este artículo a introducir será un par `(String,String)` del código *ISO 639-1* del idioma y su título. La utilidad que se le dará a esta función será la de conseguir los datos del artículo del que se buscarán recomendaciones e introducirlos en la base de datos.

Código 5.33: Función `insertarArticulo`

```
1  insertarArticulo :: (String,String) -> IO ()
2  insertarArticulo articulo = do
3     insertarTextoEnBaseDatosArticulo articulo "introsArticulos.sql"
4                                               "tablaArticulos"
```

Código 5.33: Función `insertarArticulo`

Lista introsTitulos

Accederemos a la tabla `tablaArticulos` de la base de datos creada para conseguir los títulos de los artículos y sus introducciones. Crearemos así una lista de pares de estructura `[(String,String)]` donde el primer elemento será el título y el segundo el texto de la introducción. Llamaremos `introsTitulos` a la función que nos aportará esta lista de los datos definitivos que usaremos en el recomendador.

Código 5.34: Función `introsTitulos`

```
1  introsTitulos :: IO [(String,String)]
2  introsTitulos = do
3     baseDatos <- buscadorBaseDatos
4                  "introsArticulos.sql" "SELECT titulo, texto FROM tablaArticulos"
5     let intros = zip (extraerColumna baseDatos 0) (extraerColumna baseDatos 1)
6     return $ HM.keys $ frecuencia intros
```

Código 5.34: Función `introsTitulos`

5.4.3. Palabras vacías

Una de las cosas que necesitábamos implementar para crear nuestro recomendador era una lista de palabras que el análisis ignoraría por su excesiva repetición y su poca aportación al análisis que se quiere hacer. Ya mencionamos que a este tipo de palabras se le daba el nombre de **palabras vacías**.

Hemos añadido algo menos de 500 de palabras de este tipo al módulo en una lista de nombre `palabrasVacias`. Esta lista contiene todo tipo de palabras comunes del español que no nos darían información importante como “el”, “un”, “arriba”, “mucho”, etc. , además de algunas palabras que son muy repetidas en el contexto de los árticulos de *Wikipedia* como “mundial”, “conocido”, “ubicado” o “nombre”.

Código 5.35: Principio lista `palabrasVacias`

```
palabrasVacias :: [String]
palabrasVacias = ["algún", "alguna", "algunas", "alguno", "algunos", "ambos",
            "empleamos", "ante", "antes", "aquel", "aquellas", "aquellos",
            "aquí", "arriba", "atrás", "bajo", "bastante", "bien", "cada",
            "cierta", "ciertas", "cierto", "ciertos", "como", "con",
```

Código 5.35: Principio lista `palabrasVacias`

5.4.4. Recomendador de títulos

Terminamos la aplicación con la función que nos dará la lista final de recomendaciones. La función `recomendadorArticulos` recibirá un `String` que será el título del que se quiere buscar recomendaciones, `titulo` y un número `n` en formato `Int` que será el número de recomendaciones que se darán. Por medio de `insertarArticulo` añadirá los datos del nuevo artículo a la base de datos creada `introsArticulos.sql` por `introsArticulos`. Posteriormente se realizará un llamamiento a la función `recomendador` del módulo homónimo. En el cálculo se han añadido dos valores de manera arbitraria, `50` que será el número de variables a las que se aplicará el *análisis de variables principales*, y `0.9` en referencia a que se buscará un análisis que explique como mínimo un 90 % de la varianza de estas 50 variables. Estos valores se pueden modificar para buscar un mayor ajuste en la solución final.

Código 5.36: Función `recomendadorArticulos`

```
recomendadorArticulos :: String -> Int -> IO [String]
recomendadorArticulos titulo n = do
  insertarArticulo ("es",titulo)
  datos <- introsTitulos
  return $ recomendador datos palabrasVacias titulo n 50 0.9
```

Código 5.36: Función `recomendadorArticulos`

CAPÍTULO 6

Conclusiones

Los métodos de aprendizaje automático nos rodean en nuestro día a día. Vemos ejemplos en todos los lados, ya sea en los vídeos recomendados de *YouTube*, en los algoritmos de distintas redes sociales, aplicaciones de reconocimiento de voz, detección de fraudes o incluso en cámaras que realizan análisis de sentimientos.

El objetivo del trabajo no es otro que introducir al lector en el razonamiento matemático que subyace a estos problemas. Durante la extensión del mismo, hemos analizado dos de estos procesos que, aunque su modo de afrontar los problemas es totalmente distinto —y uno de ellos carece de un conjunto de entrenamiento como tal—, hemos visto que comparten varias similitudes. De hecho en nuestro ejemplo de aplicación de un sistema no supervisado, el recomendador de artículos, sí que se utilizaba una especie de base de datos preconcebida a la que se le iban uniendo los nuevos objetivos a analizar. La principal diferencia era la no existencia de clases preconcebidas, en el aprendizaje supervisado partíamos de una clases ya establecidas; en nuestro ejemplo, los idiomas.

Ambos problemas serían adaptables a un proceso de la naturaleza opuesta, se podría hacer un clasificador de idiomas con aprendizaje no supervisado donde las componentes principales terminarían siendo los distintos idiomas, no se introducirían explícitamente, tendríamos tantos idiomas para clasificar como hubiera en la base de datos, y la respuesta final del idioma asignado vendría dada por la mayor componente principal de cada caso. De manera similar, el otro caso también es transmutable, podríamos definir ciertas categorías para cada artículo y la aplicación recomendaría una categoría de artículo.

La implementación de las librerías en el cuarto capítulo nos dejó también cosas interesantes. Pudimos ver una combinación entre el aprendizaje automático y un lenguaje de programación funcional, donde las funciones actuaban como valores y datos, y tomaban un papel principal. Además de ser una perfecta puerta de entrada al mundo de la programación desde una perspectiva matemática. Se consideraría interesante ampliar el campo de estudio del trabajo a algoritmos o aplicaciones que pudieran tener como respuesta una solución gráfica para mostrar las mejoras en la visualización y síntesis de la información de los datos. Sobre todo en el caso del análisis de componentes principales donde se pudiera reducir el número de componentes a dos o tres para poder expresar cada componente en un eje de manera sencilla, y poder darnos cuenta de la verdadera potencia de análisis que tiene este proceso.

Concluimos el trabajo habiendo explicado las claves principales de algunas de las técni-

cas más conocidas del análisis de datos referentes al aprendizaje automático, y dando algunos ejemplos de las mismas. Esperamos que los conocimientos adquiridos en este trabajo sirvan como cimientos para futuras investigaciones y proyectos.

Bibliografía

[1] James Church *Learning Haskell Data Analysis* Packt Publishing, 2015.

[2] Francisco Montes Suay *Introducción a la probabilidad* Apuntes, Universitat de València, 2007.

[3] William Feller *An Introduction to Probability Theory and Its Applications* Wiley, 1957.

[4] Roger A. Horn & Charles R. Johnson *Matrix Analysis* Cambridge University Press, 2013.

[5] Alexander McFarlane Mood, Franklin A. Graybill, Duane C. Boes *Introduction to the Theory of Statistics* McGraw-Hill, tercera edición, 1974.

[6] Ian T. Jolliffe & Jorge Cadima *Principal component analysis: a review and recent developments* The Royal Society Publishing, 2016. Disponible en: `https://royalsocietypublishing.org/doi/epdf/10.1098/rsta.2015.0202`

[7] Ionos *¿Qué es Haskell?* Página web, consultada el 11 de septiembre de 2024. Disponible en: `https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-haskell/`

[8] Analytics Vidhya *Introduction to the Powerful Bayes Theorem for Data Science* Página web, consultada el 11 de septiembre de 2024. Disponible en: `https://www.analyticsvidhya.com/blog/2019/06/introduction-powerful-bayes-theorem-data-science/?utm_source=Backlink&utm_medium=SEO`

[9] Analytics Vidhya *Naïve Bayes Algorithm* Página web, consultada el 11 de septiembre de 2024. Disponible en: `https://medium.com/analytics-vidhya/na%C3%AFve-bayes-algorithm-5bf31e9032a2`

[10] Carlos D. Carpio *Métodos de Clasificación: Naive Bayes* Página web, consultada el 12 de septiembre de 2024. Disponible en: `https://dcain.etsin.upm.es/~carlos/bookAA/02.1_MetodosdeClasificacion-Naive-Bayes.html`

[11] Wikipedia contributors *List of Wikipedias* Wikipedia, consultada el 13 de septiembre de 2024. Disponible en: `https://en.wikipedia.org/wiki/List_of_Wikipedias`

[12] Wikipedia contributors *Wikipedia* Wikipedia en español, consultada el 13 de septiembre de 2024. Disponible en: `https://es.wikipedia.org/wiki/Wikipedia`

[13] Ivan Lokhov *Wikipedia articles written by a bot* Datawrapper Blog, consultada el 13 de septiembre de 2024. Disponible en: `https://blog.datawrapper.de/wikipedia-articles-written-by-a-bot/`

[14] Joshua New *Why Do People Still Think Data Is the New Oil?* Center for Data Innovation, consultada el 27 de septiembre de 2024. Disponible en: `https://datainnovation.org/2018/01/why-do-people-still-think-data-is-the-new-oil/`

Anexo I: Salidas del clasificador Naive Bayes

En este anexo se muestran las salidas obtenidas del ejemplo de implementación del clasificador Naive Bayes en Haskell presentado con anterioridad, utilizando el entorno interactivo GHCi (Glasgow Haskell Compiler interactive).

```
ghci> :l ClasificadorIdioma
[1 of 5] Compiling Clasificador  ( Clasificador.hs, interpreted )
[2 of 5] Compiling GeneradorEnlaces ( GeneradorEnlaces.hs, interpreted )
[3 of 5] Compiling GeneradorTextos ( GeneradorTextos.hs, interpreted )
[4 of 5] Compiling GeneradorBaseDatos ( GeneradorBaseDatos.hs, interpreted )
[5 of 5] Compiling ClasificadorIdioma ( ClasificadorIdioma.hs, interpreted )
Ok, five modules loaded.
(0.79 secs,)
ghci> introsIdiomas 50
Base de datos creada con éxito.
Se han insertado 1500 elementos en la base de datos con éxito.
(1109.29 secs, 16,529,818,528 bytes)
```

La base de datos estará nutrida de 1500 introducciones de artículos aleatorios de *Wikipedia*, 50 de cada uno de los 30 idiomas. En nuestro caso, tardará 1109,29 segundos en su construcción, unos 18 minutos y medio.

Incluiremos como ejemplos la evaluación de la frase "Me alegro mucho de verte" en distintos idiomas: español, inglés, portugués, alemán, polaco, italiano y noruego.

```
ghci> clasificadorIdioma "Me alegro mucho de verte"
"es"
(1.85 secs, 1,328,026,208 bytes)
ghci> clasificadorIdiomaTodos "Me alegro mucho de verte"
[("es",1.4385796966713727e-23),("fr",2.0575862825810786e-24),
("pt",1.5006988488188862e-24),("en",1.3350031668054959e-24),
("nl",4.312475639514897e-25),("ca",1.868594793261105e-25),
("sv",7.40659682333777e-26),("ro",7.234569525937924e-26),
("it",2.457068448809503e-26),("tr",2.0575288642321292e-26),
("hu",1.666880121158094 6e-26),("no",1.1000231787011853e-26),
("de",1.0814625409986618e-26),("ar",9.750177632771201e-27),
("ja",9.45650958601682e-27),("id",6.475369149049737e-27),
("fa",6.467582829508386e-27),("uk",5.923137433434641e-27),
("ru",5.703296873969729e-27),("pl",5.6597861219381146e-27),
("zh",5.320793439211851e-27),("vi",4.238956125487497e-27),
```

```
("he",4.1435008947009256e-27),("el",2.0638316732414866e-27),
("th",1.8556563972317497e-27),("ko",1.3629206500101532e-27),
("cs",1.2430390810619446e-27),("fi",9.051011209785674e-28),
("bn",7.818284158264162e-28),("hi",4.716927727546853e-28)]
(1.97 secs, 1,329,142,192 bytes)
ghci> clasificadorIdioma "I'm very happy to see you"
"en"
(2.09 secs, 1,522,266,088 bytes)
ghci> clasificadorIdioma "Estou muito feliz em ver você"
"pt"
(2.06 secs, 1,522,285,544 bytes)
ghci> clasificadorIdioma "Ich freue mich sehr, Sie zu sehen"
"de"
(2.28 secs, 1,716,540,048 bytes)
ghci> clasificadorIdioma "Bardzo się cieszę, że cię widzę"
"pl"
(2.92 secs, 1,522,319,184 bytes)
ghci> clasificadorIdioma "Sono molto felice di vederti"
"it"
(1.95 secs, 1,328,045,632 bytes)
ghci> clasificadorIdioma "Jeg er veldig glad for åse deg"
"no"
(2.61 secs, 1,910,765,376 bytes)
```

A pesar de tratarse de un ejemplo muy simple con una frase de pocas palabras, podemos comprobar el buen funcionamiento de nuestro clasificador, asignando el idioma correcto en cada caso.

Anexo II: Salidas del recomendador basado en el Análisis de Componentes Principales

En este anexo se incluyen ejemplos de algunas salidas generadas por el recomendador basado en el Análisis de Componentes Principales, desarrollado durante este trabajo, implementado en Haskell y ejecutado en el entorno interactivo GHCi (Glasgow Haskell Compiler interactive).

```
ghci> :l RecomendadorArticulos
[1 of 5] Compiling GeneradorEnlaces ( GeneradorEnlaces.hs, interpreted )
[2 of 5] Compiling GeneradorTextos ( GeneradorTextos.hs, interpreted )
[3 of 5] Compiling GeneradorBaseDatos ( GeneradorBaseDatos.hs, interpreted )
[4 of 5] Compiling Recomendador ( Recomendador.hs, interpreted )
[5 of 5] Compiling RecomendadorArticulos ( RecomendadorArticulos.hs, interpreted
    )
Ok, five modules loaded.
(1.14 secs,)
ghci> introsArticulos 1500 "es"
Base de datos creada con éxito.
Se han insertado 1500 elementos en la base de datos con éxito.
(929.72 secs, 16,246,341,576 bytes)
```

La base de datos se conformará de 1500 introducciones de artículos aleatorios de *Wikipedia* en español. En este caso, su creación se ejecuta en 929,72 segundos, aproximadamente 15 minutos y medio.

Incluiremos ejemplos como: *Usain Bolt*, *Sevilla*, Manzana, *Albert Einstein* y *España*.

```
ghci> recomendadorArticulos "Usain Bolt" 10
Insertado en la base de datos con éxito.
["AlphaGo Zero","Serie Nacional de B\233isbol 1965-1966","Adri\225n Garc\237a","
    Stefania Belmondo","Tatsuhiro Yonemitsu","Indonesia en los Juegos Ol\237
    mpicos de Roma 1960","Mauritania en los Juegos Paral\237mpicos de S\237dney
    2000","Mal\237 en los Juegos Ol\237mpicos de Londres 2012","Miguel Martinez (
    ciclista)","Lilou Llu\237s Valette"]
(2.24 secs, 743,440,056 bytes)
ghci> recomendadorArticulos "Sevilla" 10
Insertado en la base de datos con éxito.
["Sitio de Ciudad Rodrigo (1707)","Sombrerete","Neckarsulm","Casino Militar de
    Melilla","La Encarnaci\243n (desambiguaci\243n)","Novoros\237isk","Dom\382ale
```

```
    ","Hoery\335ng","\346winouj\347cie","Circunscripci\243n electoral de Zaragoza
    "]
(3.24 secs, 737,889,256 bytes)
ghci> recomendadorArticulos "Manzana" 10
Insertado en la base de datos con éxito.
["Phalonidia lydiae","Messor reticuliventris","Derolus griseonotatus","
    Choristoneura occidentalis","Eucalyptus regnans","Nyssodrysternum fulminans
    ","Phalonidia docilis","Brancasaurus","Strangalia beltii","Xixuthrus
    terribilis"]
(2.24 secs, 743,213,920 bytes)
ghci> recomendadorArticulos "Albert Einstein" 10
Insertado en la base de datos con éxito.
["Descubrimientos de pies humanos en el Mar de los Salish","Maeve Brennan","
    Andrew N. Dugger","Shave and a Haircut","John Smith (luchador ol\237mpico)","
    Pr\243tesis de retina Argus","Atlantismo","Transporte Presidente Pinto","Isla
     de hielo de Fletcher","Samantha Arsenault"]
(2.52 secs, 746,537,408 bytes)
ghci> recomendadorArticulos "España" 10
Insertado en la base de datos con éxito.
["\205ndice de referencia de pr\233stamos hipotecarios","C\233sar E. Arroyo","
    Invasi\243n de Portugal (1807)","Jos\233 Gautier Ben\237tez","Comisiones
    Obreras de Euskadi","Circunscripci\243n electoral de Zaragoza","Condado del
    Puente","Menorca (gallina)","Archivo Hist\243rico Minero de la Fundaci\243n R
    \237o Tinto","Boeng Krum"]
(2.30 secs, 753,417,248 bytes)
```

En general, podemos observar cómo se relacionan::

- *Usain Bolt* con nombres de deportistas y eventos deportivos
- *Sevilla* con otras ciudades o artículos relacionados
- *Manzana* con nombres científicos propios de la biología
- *Albert Einstein* con artículos vinculados al conocimiento
- *España* con artículos asociados a España