TRABAJO FIN DE GRADO

Doble Grado Ingeniería Informática y Matemáticas

Detección de anomalías basada en técnicas de ensembles

Biblioteca de algoritmos

Autor
Ignacio Aguilera Martos

Directores
Francisco Herrera Triguero

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Facultad de Ciencias

—

Granada, 5 de Septiembre de 2019

Detección de anomalías basada en técnicas de ensembles

Biblioteca de algoritmos

Autor
Ignacio Aguilera Martos

Directores
Francisco Herrera Triguero

Detección de anomalías basada en técnicas de ensembles: Biblioteca de algoritmos

Ignacio Aguilera Martos

Palabras clave: outlier, anomalía, ensemble, python

Resumen

Poner aquí el resumen.

Outlier detection based in ensemble methods: Library implementation

Ignacio Aguilera Martos

Keywords: outlier, ensemble, anomaly, python

Abstract

Write here the abstract in English.

Agradecimientos

Poner aquí agradecimientos...

Índice general

1. Introducción **1**
1.1. Contextualización 2

I Machine Learning **5**

2. Machine Learning **7**
2.1. Contextualización del aprendizaje 7
2.1.1. Objetivo del aprendizaje 8
2.1.2. Clases de aprendizaje 9
2.2. Principios y adaptación del aprendizaje 10
2.2.1. Principios inductivos 12
2.3. Regularización 15
2.3.1. Problema de la alta dimensionalidad 15
2.3.2. Aproximación de funciones 18
2.3.3. Penalización o control de la complejidad 19
2.3.4. Equilibrio entre el sesgo y la varianza 21
2.4. Teoría estadística del aprendizaje 24
2.4.1. Condiciones para la convergencia y consistencia del ERM 24
2.4.2. Función de crecimiento y dimensión de Vapnik-Chervonenkis 27
2.4.3. Límites de la generalización 30
2.4.4. Principio de minimización del error estructural (SRM) 32
2.4.5. Aproximaciones de la dimensión VC 33
2.4.6. Perspectiva 34

3. Concepto de anomalía **35**
3.1. Contextualización 35
3.2. Criterios 35
3.3. Qué hacer con las anomalías 37

4. Introducción de Estadística Multivariante **39**
4.1. Introducción 39
4.1.1. Independencia 41

4.1.2. Probabilidad y esperanza condicionada 42
4.1.3. Desigualdades y fórmulas famosas 47

5. Concepto probabilístico de anomalía **51**

6. Modelos implementados **55**
6.1. Algoritmos de ensamblaje 55
6.2. Mahalanobis Kernel 56
6.3. TRINITY 59
6.4. OUTRES 63
6.5. HICS 71
6.6. LODA 78
6.7. Implementación 82

Bibliografía **88**

Capítulo 1

Introducción

Antes de comenzar el objeto de estudio de este trabajo, lo primero que debemos hacer es contextualizar el mismo y establecer un marco de trabajo en cuanto a teoría que se empleará en la posterior experimentación y desarrollo del mismo.

El estudio realizado y plasmado en este trabajo se centra en la obtención de técnicas para la detección de anomalías en conjuntos de datos, concepto que introduciremos posteriormente. En concreto las técnicas que se van a desarrollar son las conocidas como técnicas de ensemble que se basan en el estudio del problema o bien combinando modelos existentes o bien haciendo un estudio pormenorizado aplicando algún criterio por subespacios del conjunto de datos.

En primer lugar el trabajo desarrollará una introducción a la teoría de aprendizaje y resolución de problemas mediante datos y no por diseño así como la teoría matemática que esto involucra. Esta primera sección nos dará suficiente estructura al trabajo para poder definir en términos de distancias lo que significa que una instancia de un conjunto de datos sea una anomalía.

Posteriormente se desarrollará brevemente algunos conocimientos estadísticos básicos de estadística multivariante para poder introducir el concepto de anomalía desde la perspectiva de las probabilidades condicionadas.

Tras esto podremos entrar en el terreno de la experimentación, desarrollo y explicación de técnicas y puesta en contraste con los algoritmos clásicos para comprobar el desempeño de las nuevas técnicas.

Por último se presentarán las conclusiones obtenidas tras todo este estudio.

1.1. Contextualización

En este contexto y sin haber comenzado la discusión del problema podríamos considerar el establecimiento de una solución al problema de detección de anomalías empleando un algoritmo preciso y óptimo, aunque costoso en tiempo, pero esta solución no es viable en este contexto. Pensemos por un momento en cómo detectar una anomalía en un conjunto de datos. Para poder dar un algoritmo concreto que resuelva siempre el problema de forma óptima tenemos que estar seguros de que somos capaces de definir de forma clara y para todo conjunto de datos cuándo estamos en presencia de un dato anómalo. Pero, ¿sabemos exactamente cuándo sobrepasamos la barrera de dato ligeramente desviado, ruido o una anomalía?

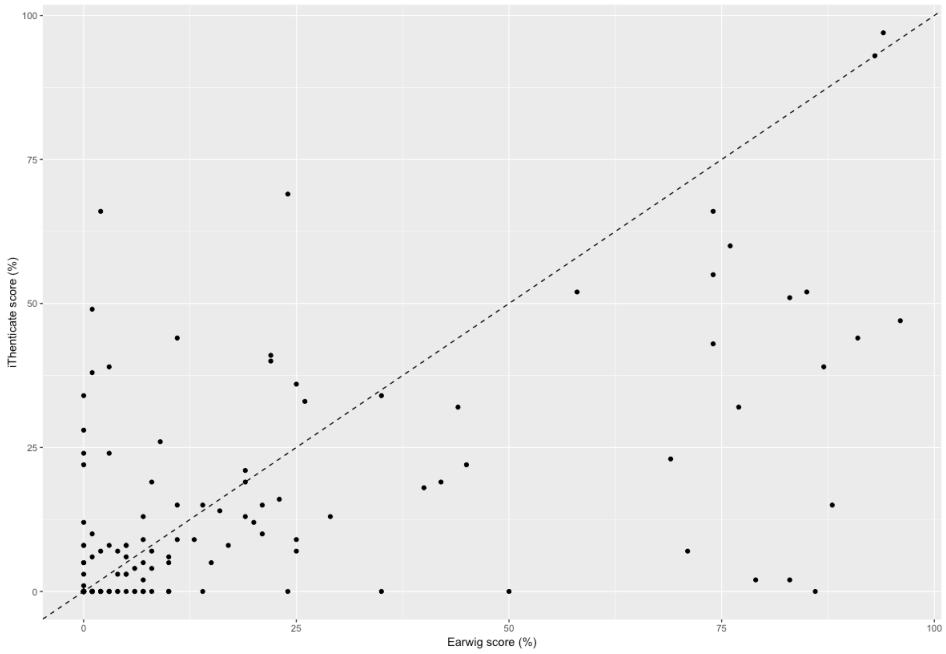

Figura 1.1: Ejemplo de conjunto de datos (Wikimedia)

Pensemos por ejemplo en este conjunto de datos. Tenemos un cluster muy bien definido en la esquina inferior izquierda y datos dispersos. ¿Podemos afirmar claramente cual es la línea que distingue los datos anómalos de los normales?

Esta discusión nos lleva a la primera barrera que tenemos que superar en este trabajo. El problema no es resoluble por diseño y por tanto debemos realizar una aproximación a través de los datos. Esto nos va a llevar a aprender de los mismos, valorar cada dato con una puntuación que nos acabará discriminando los datos normales y anómalos.

En el contexto de los problemas resueltos a través de los datos tenemos dos tipos: clasificación y regresión.

Podríamos pensar que estamos ante un problema de clasificación ordinario, es decir, aprender a través de los datos de entrenamiento cuáles son las anomalías y cuales los datos normales pero no es así. Pensemos que no sabemos ni siquiera nosotros clasificar claramente estas anomalías salvo en algunos ejemplos prácticos. Por ejemplo si le diéramos a una persona un conjunto de datos ya establecido como el que hemos visto anteriormente 1.1 no sería nada fácil e incluso no podemos afirmar que sepamos hacer esta tarea. Por contra si el conjunto de datos es generado en un proceso de trabajo, por ejemplo un conductor de camiones haciendo su ruta, podemos saber cuándo hay una anomalía porque el trabajador la está viviendo y puede señalarla.

Por tanto este problema no es de clasificación al uso pues no es un problema supervisado. Esto significa que, en general no vamos a tener un conjunto etiquetado en el que poder probar cómo aprende nuestro modelo y tendremos que abordar otras técnicas para comprobar el desempeño.

Ahora que comprendemos el alcance del problema podemos intentar profundizar un poco más en la base teórica del mismo analizando las herramientas que el machine learning nos provee para abordarlo.

Parte I

Machine Learning

Capítulo 2

Machine Learning

En este capítulo vamos a hacer un repaso sobre los conceptos asociados al Machine Learning, el aprendizaje y la teoría matemática que involucra. Estas herramientas y conceptos los utilizaremos posteriormente para resolver el problema de detección de anomalías.

2.1. Contextualización del aprendizaje

Para comenzar tenemos que empezar definiendo en que consiste el proceso de aprender sobre unos datos. Supongamos que tenemos un problema en el que tenemos una entrada y una salida, por ejemplo una entrada válida podría ser un vector $x \in \mathbb{R}^d$ y una salida un valor real o un número natural. El problema de aprendizaje intenta estimar una estructura de tipo entrada-salida como la descrita usando únicamente un número finito de observaciones.

Podemos definirlo de forma más general empleando tres conceptos:

- Generador: El generador se encarga de obtener las entradas $x \in \mathbb{R}^d$ mediante una distribución de probabilidad $p(x)$ desconocida y fijada de antemano.
- Sistema: El sistema es el que produce la salida "y" (correcta) para cada entrada $x \in \mathbb{R}^d$ mediante la distribución de probabilidad $p(x|y)$ desconocida y fijada de antemano.
- Máquina de aprendizaje: esta es la que va a obtener información de las entradas y salidas conocidas para intentar predecir la salida co-

rrecta para una entrada nueva que se nos de. De forma abstracta esta máquina lo que hace es tomar una serie de funciones de un conjunto general de forma que para una entrada dada x la función $f(x, \omega)$ con $\omega \in \Omega$ nos de la salida que corresponde para x donde ω es una forma de indexar las funciones tomadas para generalizar la salida del conjunto más general de funciones que hemos indicado.

El único cabo que hemos dejado sin atar en las definiciones que acabamos de ver es el conjunto de funciones del cual tomaremos algunas para adaptar la máquina de aprendizaje a los datos recibidos. Este conjunto de funciones, que notaremos como $\mathcal{H}$, es de momento la única forma que tenemos de aplicar un conocimiento a priori en la máquina de aprendizaje.

Para finalizar esta breve introducción y poder continuar profundizando vamos a exponer algunos ejemplos de clases de funciones para que podamos visualizar el contexto.

- Funciones lineales: En este caso la clase de funciones $\mathcal{H}$ está formada por funciones de la forma $h(x) = w_0 + \sum_{i=1}^{d} x_i w_i$ donde $w \in \mathbb{R}^{d+1}$. Este es el modelo de funciones más clásico.
- Funciones trigonométricas: Un ejemplo de una clase de funciones trigonométricas podría ser $f_m(x, v_m, w_m) = \sum_{j=1}^{m-1}(v_j \sin(jx) + w_j \cos(jx)) + w_0$ donde en este caso la entrada es un único valor real. Este tipo de clases de funciones serán útiles en problemas de regresión que luego explicaremos con algo más de detalle aunque sin centrarnos mucho en ello pues no es el objetivo del estudio.

2.1.1. Objetivo del aprendizaje

Cuando hablamos de aprendizaje queremos obtener algo de dicho aprendizaje a partir de los datos. Como ya se ha mencionado, intentamos obtener una función de una familia de funciones que aproxime o modele de buena manera la salida del sistema. Por tanto, ese es nuestro objetivo: obtener una función de la familia de funciones que minimice el error.

El problema que enfrentamos es que sólo disponemos de un número finito, por ejemplo n, de observaciones de datos y su correspondiente salida. Esto nos va a hacer que no podamos tener una garantía de optimalidad a no ser que tendamos n a infinito.

Sin embargo si que podemos cuantificar cómo de buena es una aproximación con respecto a otra mediante la función pérdida o error que denotaremos

como $L(y, f(x, \omega))$. Esta función nos va a medir la diferencia entre la salida real del sistema y la salida dada por la función f para la entrada x siendo siempre $L(y, f(x, \omega)) \geq 0$.

Recordemos además que el Generador obtiene datos mediante una distribución desconocida pero fijada de antemano y que son independientes e idénticamente distribuidos con respecto a la distribución conjunta, es decir:

$$p(x, y) = p(x)p(y|x)$$

Una vez definido todo esto podemos obtener el valor esperado de pérdida o error mediante el funcional

$$R(\omega) = \int L(y, f(x, \omega))p(x, y)dxdy$$

Ahora podemos concretar un poco más lo que entendemos como objetivo del aprendizaje. El objetivo será encontrar una función $f \in \mathcal{H}$ que nos minimice el valor del funcional $R(\omega)$. Pero recordemos que $p(x, y)$ es desconocida para nosotros, por lo que no podemos saber cómo se distribuyen los datos y por tanto el valor del funcional no es calculable para nosotros y por tanto la solución puramente de cálculo no es accesible.

Por tanto, la única forma realmente potente y útil de encontrar una buena aproximación será incorporar el conocimiento a priori que tenemos del sistema. En la sección anterior hemos visto que una forma de incorporar dicho conocimiento es mediante la selección de la clase de funciones, pero además será muy relevante el hecho de cómo los datos son empleados en el proceso de aprendizaje. En este apartado de decisión tendremos que resolver primero la codificación de los datos, el algoritmo empleado y el uso de técnicas como la regularización que veremos después para incorporar nuestro conocimiento en el camino que nos lleve a la solución.

2.1.2. Clases de aprendizaje

El problema de aprendizaje puede ser subdividido a su vez en cuatro clases distintas y que se suelen abordar de forma independiente. Estoss tipos de problemas de aprendizaje son:

- Clasificación: El problema de clasificación consiste en identificar y separar instancias de datos según su clase. Por ejemplo podemos dividir a

la población mundial en dos clases: sanos y enfermos. Un problema de clasificación podría ser saber identificar estas clases para un conjunto de personas. Los problemas de clasificación más sencillos son aquellos en los que se usan dos únicas clases aunque se puede generalizar la definición del problema a k-clases.

- Regresión: El problema de regresión consiste en estimar una función $f : \mathbb{R}^n \to \mathbb{R}$ a partir de una serie de muestras previas con los valores de f. Un problema de regresión podría ser determinar la función que, dados los datos de altura y dimensiones corporales sea capaz de darnos el peso aproximado de la persona.

- Estimación de la función de densidad: en este caso no nos interesa la salida que proporciona el sistema, ya sea el valor de una clase o una función real como en el caso de la regresión. En este caso el objetivo del aprendizaje es conseguir la función de densidad $f(x, \omega)$, con $\omega \in \Omega$ los parámetros necesarios de la función de densidad, con la que se distribuyen los datos de entrada del sistema.

- Agrupamiento y cuantificación vectorial: El problema de cuantificación vectorial consiste en intentar explicar la distribución de los vectores de entrada mediante puntos clave llamados centroides. De esta forma se podría reducir la complejidad de los datos expresándolos en función de un sistema de generadores menor. El problema de agrupamiento tiene también relación por utilizar la idea de centroide, pero el objetivo es completamente distinto. El objetivo del problema de agrupamiento es intentar conseguir agrupar los datos en clusters, es decir, regiones del espacio en las que se concentran un conjunto de datos. De esta forma intentamos agrupar los datos que mantienen una relación entre sí. Un ejemplo de un problema de cuantificación vectorial podría ser un problema de reducción de dimensionalidad y un ejemplo de problema de agrupamiento podría ser identificar instancias de datos con características comunes.

2.2. Principios y adaptación del aprendizaje

Según Vapnik [1] la predicción mediante el aprendizaje se puede dividir en dos fases:

1. Aprendizaje o estimación a partir de una muestra.

2. Predicción a partir de las estimaciones obtenidas.

Estas dos fases se corresponden con los dos tipos de inferencia clásica que conocemos, esto es, inducción y deducción. Traído a este caso el proceso de inducción es aquel que a partir de los datos de aprendizaje o los datos de la muestra que tenemos con la salida que corresponde podemos estimar un modelo. Es decir, estamos sacando el conocimiento de los datos para generar el modelo. El proceso de deducción es aquel que, una vez obtenido el modelo estimado (la generalización) obtenemos una predicción de la salida sobre un conjunto de datos.

Por contra, Vapnik propone un paso que resuelve estas dos fases directamente y que él denomina transducción. Este paso consiste en, dados los datos de entrenamiento obtenemos directamente los valores de salida sin tener que hacer la generalización a un modelo. De esta forma, según Vapnik, podríamos reducir el error que cometemos en la predicción. Este razonamiento tiene sentido, pues estamos omitiendo el paso más complejo del proceso de inducción-deducción.

En resumen esta idea se puede resumir en la siguiente figura:

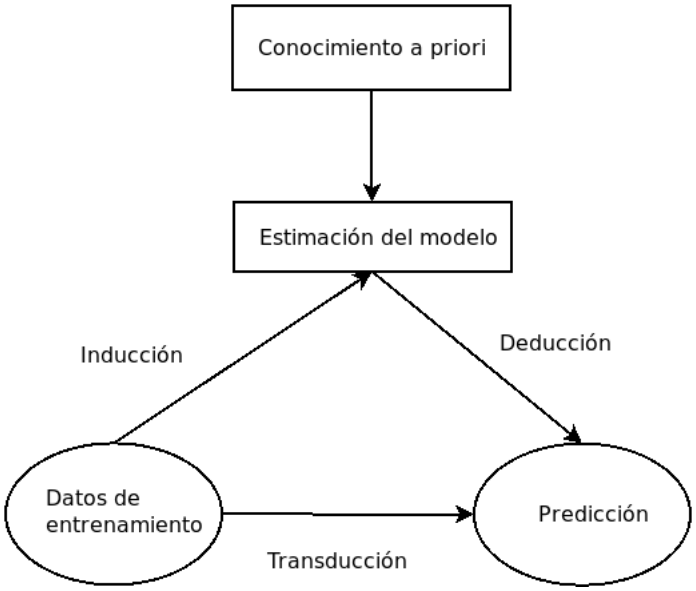

Figura 2.1: Tipos de inferencia y transduccion [2, p. 41]

Podemos ver que el conocimiento a priori que tenemos del problema se manifiesta una vez se crea el modelo general, de forma que se emplearía en el paso de la inducción. Ya hemos hablado previamente del conocimiento a priori y cómo incorporarlo al modelo, pero por concretar un poco más podemos añadirlo básicamente de dos formas:

- Escogiendo un conjunto de funciones para aproximar la salida del sistema
- Añadiendo restricciones o penalizaciones adicionales a dicho conjunto de funciones.

En resumen, para poder crear la generalización del modelo de forma única necesitamos:

1. Un conjunto de funciones para aproximar la salida.
2. Conocimiento a priori.
3. Un principio inductivo, que no es más que una indicación de cómo emplear los datos para llegar a la generalización del modelo.
4. Un método de aprendizaje, es decir, una implementación del principio inductivo.

En secciones posteriores revisaremos algunos de los principios inductivos más usados pero es importante reseñar la diferencia entre pricipio inductivo y método de aprendizaje. Para un mismo principio inductivo podemos tener varios métodos de aprendizaje, pues podemos escoger diferentes formas de llevarlo a la práctica. Por ejemplo, uno de los principios inductivos más empleados es el ERM o Empirical Risk Minimization, es decir, minimización del error empírico. Podríamos pensar en diferentes formas de utilizar este principio, por ejemplo sólo avanzamos en la creación del modelo si a cada paso que demos minimizamos el error, o por ejemplo vamos avanzando varios modelos a la vez hasta obtener un número de modelos finales de entre los cuales escogeremos aquel que mejor minimice dicho error.

2.2.1. Principios inductivos

Una vez introducido el concepto como hemos hecho en la sección anterior vamos a hacer un breve repaso de los principios más usados y en qué consiste cada uno de ellos.

Penalización o Regularización

Imaginemos que tenemos una clase de funciones muy flexible, esto es con un gran número de parámetros libres $f(x, \omega)$ con $\omega \in \Omega$. Vamos a partir de

la base del ERM, es decir, minimizar el error empírico. La penalización lo que va a hacer es añadir un factor a la función a minimizar:

$$R_{pen}(\omega) = R_{emp}(\omega) + \lambda\phi[f(x,\omega)]$$

Donde $R_{emp}(\omega)$ es el error empírico con los parámetros ω y $\phi[f(x,\omega)]$ es un funcional no negativo asociado a cada estimación $f(x,\omega)$. El parámetro $\lambda > 0$ es un escalar que controla el peso de la penalización.

El funcional $\phi[f(x,\omega)]$ puede medir lo que creamos conveniente que debemos añadir, es decir, aquí podemos añadir a la minimización algún tipo de medida que nos diga cómo de bien funciona el ajuste de los datos y cómo de bien funciona la información a priori que hemos incluido en el modelo. Pensemos por ejemplo que λ fuera un parámetro con un valor muy alto. En este caso la penalización por un mal ajuste de los datos no sería de gran importancia pues lo más conveniente sería minimizar el valor del funcional para no obtener una gran penalización. De esta forma podemos ajustar y dar un poco más de información al error empírico. Por ejemplo, en función del problema, es posible medir la complejidad de la solución mediante el funcional ϕ y de esta forma no sólo vamos a obtener una función que ajuste bien los datos, si no que también mantenga una cierta simplicidad para evitar por ejemplo el sobreajuste.

Reglas de parada anticipada

Pensemos en un método que vaya aprendiendo de los datos de forma iterativa intentando a cada iteración reducir el error cometido, por ejemplo el ERM. Los métodos o reglas de parada anticipada pueden verse como penalizaciones sobre el algoritmo conforme se va ejecutando. Las reglas de parada anticipada, como su nombre indica lo que preveen es la parada del algoritmo antes de obtener su objetivo teórico. Por ejemplo un algoritmo intenta que el error sea menor que 10^{-6} pero para reducirlo desde 10^{-4} hasta 10^{-5} está consumiendo millones de iteraciones. Si queremos que el tiempo de cómputo penalice lo que podemos hacer es fijar por ejemplo un número máximo de iteraciones que detenga el método aunque no se haya alcanzado esa barrera de error que se preveía.

Minimización del riesgo estructural o SRM

Para entender esta filosofía nos ponemos en la situación de que ya sabemos la clase de funciones con la que vamos a aproximar la salida del sistema,

por ejemplo hemos escogido la clase de funciones polinómicas. Bajo esta clase de funciones podemos ordenar las funciones por complejidad, entendiendo por complejidad el número de parámetros de la función. Por ejemplo los polinomios de grado m son de menor complejidad que los de grado $m+1$. De esta forma podemos pensar en una estructura de la clase de funciones de la forma:

$$S_0 \subset S_1 \subset S_2 \subset \cdots$$

Este parámetro de complejidad también puede ser un principio a minimizar para intentar conseguir una solución adecuada pero también simple. La generalización de la medida de complejidad para las clases de funciones es la conocida como dimensión VC o dimensión de Vapnik-Chervonenkis.

Inferencia Bayesiana

Este principio inductivo se utiliza en el problema de estimación de la función de densidad. El principio es utilizar la conocida fórmula de Bayes para hacer una estimación de la función de densidad empleando el conocimiento a priori que disponemos del problema. La forma en la que se emplea esta fórmula es de la siguiente:

$$P[modelo|datos] = \frac{P[datos|modelo] \cdot P[modelo]}{P[datos]}$$

donde $P[modelo]$ es la probabilidad a priori, $P[datos]$ es la probabilidad de los datos de entrenamiento y $P[datos|modelo]$ es la probabilidad de que los datos estén generados por el modelo.

Descripción de mínima longitud

La idea de este principio es la minimización de la longitud que se necesita emplear para describir un modelo y la correspondiente salida. Llamamos l a la longitud total:

$$l = L(modelo) + L(datos|modelo)$$

Esta medida puede ser vista como una medida de complejidad conjunta de todo el modelo.

2.3. Regularización

Por la importancia de este principio inductivo vamos a desarrollarlo un poco más, junto con el concepto de penalización, la selección de los modelos y la relación entre sesgo y varianza. Este último es un concepto muy relevante en cuanto al aprendizaje y que en nuestro caso, al no poseer la clasificación real tendremos que tenerlo en cuenta.

2.3.1. Problema de la alta dimensionalidad

Sabemos que cuando estamos ante un problema de aprendizaje nuestro objetivo es conseguir estimar una función con un número finito de instancias de una muestra ya con la salida. Al tener un número finito de elementos en la muestra ya sabemos que no podemos garantizar que la respuesta sea la óptima o correcta, pero además debemos pensar que a mayor regularidad del conjunto de funciones empleado debemos tener una densidad suficiente de puntos para compensar dicha regularidad. Este problema es conocido como la maldición de la dimensionalidad (curse of dimensionality). El problema es que cuanto mayor sea la dimensionalidad considerada más difícil es poder tener esa alta densidad de datos que se requieren para funciones muy regulares.

Este problema que conlleva la alta dimensionalidad proviene de la geometría de los espacio con alta dimensionalidad. A medida que incrementamos la dimensionalidad el espacio se ve cada vez con más aristas o picos. Podemos pensar en un cubo para el espacio tridimensional y a medida que aumentamos la dimensión incorporamos más aristas y vértices. Podemos resumir en 4 propiedades de los espacio con alta dimensionalidad que causan este problema:

1. La densidad disminuye exponencialmente al aumentar el número de dimensiones. Supongamos que tenemos una muestra de n puntos en $\mathbb{R}$. Para poder tener la misma densidad en un espacio d-dimensional $\mathbb{R}^d$ necesidamos n^d puntos.

2. Cuanto mayor dimensionalidad tenga el conjunto de datos mayor lado se necesita para que un hipercubo contenga el mismo porcentaje del conjunto que con una menor dimensionalidad. Imaginemos que tenemos un conjunto d-dimensional en el que tenemos la muestra dentro de un hipercubo unidad. Si quisiéramos abarcar un porcentaje $p \in [0, 1]$ necesitaríamos un cubo de lado $e_d(p) = p^{\frac{1}{d}}$. Como se puede observar a

mayor dimensionalidad y p constante el lado es cada vez mayor. Esta idea es fácilemente entendible si observamos la siguiente figura:

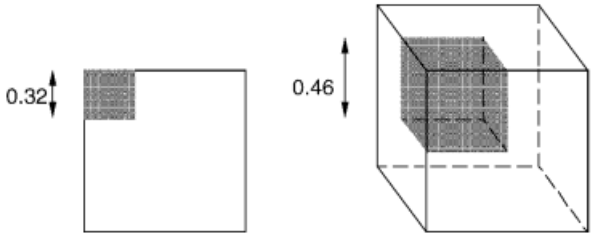

Figura 2.2: Para 2 dimensiones necesitamos menor lado que para 3 dimensiones. [2, p. 64]

3. Casi todo punto está más cerca de un borde que de otro punto. Pensemos en un conjunto de datos con n puntos distribuidos de forma uniforme en una bola d-dimensional de radio unidad. Para este conjunto de datos, según [3], la distancia media entre el centro de la distribución y los puntos más cercanos a dicho centro se mide bajo la fórmula:

$$D(d,n) = (1 - \frac{1}{2}^{1/n})^{1/d}$$

Si en esta fórmula tomamos por ejemplo $n = 200$ y $d = 10$ el resultado es $D(10, 200) \approx 0.57$. Esto significa que los puntos más cercanos al centro de la distribución están más cerca de los bordes que del centro.

4. Casi todo punto es una anomalía sobre su propia proyección. Si pensamos de nuevo en la idea de los vértices y aristas en espacio de alta dimensionalidad y pensamos en que, según el punto anterior, cada vez que aumenta la dimensionalidad los puntos están más cerca de los bordes entonces no es extraño pensar que los puntos a medida que aumenta la dimensionalidad están más distantes del resto de puntos. Esto intuitivamente (ya que aún no hemos visto la definición formal de anomalía) nos guía a pensar que vistos los puntos en sus propios entornos éstos serán anomalías comparados con el resto.

Figura 2.3: Forma conceptual de un espacio de alta dimensionalidad.[2, p. 64]

> Conceptualmente podemos imaginarlo con esta forma de picos, con lo que si tenemos los datos apiñados en dichos picos o extremos el resto de datos que estén en picos diferentes distan tanto del que estamos considerando que no podemos afirmar que tengan ninguna relación entre sí.

Estos puntos hemos de recordar que van referidos al conjunto de datos y no a las funciones que estamos considerando para representar la salida del sistema. Si estamos considerando la complejidad de las funciones la dimensionalidad no es una buena medida. Sabemos de la existencia de teoremas de aproximación de funciones como por ejemplo el Teorema de Superposición de Kolmogorov-Arnold.

Teorema 2.1 (Teorema de Superposición de Kolmogorov-Arnold) *Sea f una función continua de varias variables $f : X_1 \times ... \times X_n \to \mathbb{R}$, entonces existen funciones $\Phi_q : \mathbb{R} \to \mathbb{R}$ y $\phi_{q,p} : X_p \to [0, 1]$ tales que f se puede expresar como:*

$$f(x) = f(x_1, ..., x_n) = \sum_{q=0}^{2n} \Phi_q(\sum_{p=1}^{n} \phi_{q,p}(x_p))$$

Este Teorema argumenta perfectamente que la complejidad que le damos a los datos por tener una alta dimensionalidad no es transferible a las funciones pues podemos expresar funciones de varias variables como combinación de funciones de una sola variables. En otras palabras no podemos argumentar que la complejidad de funciones univariantes sea mayor o menor que la de funciones multivariantes.

2.3.2. Aproximación de funciones

Como ya hemos dicho en la introducción queremos aproximar una función salida del sistema dentro de una familia de funciones. Este campo no es nuevo, tenemos como herramientas una serie de Teoremas relacionados con la aproximación de funciones como el Teorema de Kolmogorov enunciado anteriormente o el Teorema de aproximación de Weierstrass.

La versión más simple del Teorema de Weierstrass es la de funciones reales definidas en intervalos cerrados, veamos un repaso de estos Teoremas para hacer un esquema de la aproximación de funciones.

Teorema 2.2 (Teorema de aproximación de Weierstrass) *Supongamos que* $f : [a, b] \to \mathbb{R}$ *es una función continua. Entonces* $\forall \epsilon > 0$, $\exists p$ *un polinomio tal que* $\forall x \in [a, b]$ *tenemos que* $|f(x) - p(x)| < \epsilon$.

En otras palabras, podemos aproximar las funciones continuas reales definidas en un intervalo cerrado con el error que queramos en un punto mediante polinomios. Además tenemos versiones más generales aún como el Teorema de Stone-Weierstrass para funciones reales, para espacios localmente compactos y para el espacio de los complejos.

Estas aproximaciones son más sencillas en términos de la complejidad de la clase de funciones, pero tenemos aproximaciones muy famosas, como por ejemplo la serie de Fourier.

Definición 2.1 (Serie de Fourier) *Si tenemos una función* $f : \mathbb{R} \to \mathbb{R}$ *integrable en el intervalo* $[t_0 - \frac{T}{2}, t_0 + \frac{T}{2}]$ *entonces se puede obtener el desarrollo en serie de Fourier de* f *en dicho intervalo. Si* f *es periódica en toda la recta real la aproximación será válida en todos los valores en los que esté definida.*

$$f(t) \approx \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(\frac{2n\pi}{T} t) + b_n \sin(\frac{2n\pi}{T} t)]$$

Donde a_0, a_n *y* b_n *son los coeficientes de la serie de Fourier que tienen la forma:*

$$a_0 = \frac{2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) dt$$

$$a_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)cos(\frac{2n\pi}{T}t)dt$$

$$b_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\sin(\frac{2n\pi}{T}t)dt$$

Como podemos ver hemos introducido dos conocidas formas de aproximar funciones, una con funciones polinómicas y otra con funciones trigonométricas. Vamos a dividir en dos los tipos de aproximación que podemos tener para el problema de aprendizaje.

1. Aproximaciones universales: son aquellas en las que se establece que cualquier función continua puede ser aproximada por otra función de otra clase con el error que queramos. En este grupo podríamos meter a los dos teoremas que hemos dado previamente. Dentro de este grupo podemos tener diferentes tipos de aproximaciones en función de la familia de funciones que escojamos como aproximaciones. Por ejemplo en los dos teoremas previos hemos cogido las clases de funciones polinómicas y trigonométricas pero podríamos haber tomado otras clases diferentes.

2. Aproximaciones inexactas: son aquellas en las que no podemos tener una aproximación como las que hemos dado en los teoremas previos, si no que proveen de una aproximación de peor calidad.

2.3.3. Penalización o control de la complejidad

Ya hemos discutido brevemente en la sección de principios inductivos la complejidad y cómo penalizarla. Vamos a ver qué elementos queremos controlar con la penalización:

1. La clase de funciones con la que vamos a hacer la aproximación. Tenemos que decidir si escoger una clase tan amplia que nos aseguremos que abarque la solución seguro pero penalicemos la complejidad de la elección o queremos una clase de funciones más ajustada.

2. Tipo de funcional de penalización. Tenemos que escoger entre los distintos tipos de penalización que queremos. Esto se reduce a escoger entre dos tipos de penalización: paramétrica y no paramétrica. La primera de ellas se basa en estudiar la suavidad del ajuste junto con el

número de parámetros que requiere la aproximación mientras que la segunda intenta estudiar lo mismo, es decir la suavidad del ajuste, sin medir los parámetros de la clase de funciones. En este punto se puede incorporar el conocimiento a priori del problema.

3. Método con el que queremos minimizar la penalización. Este apartado está relacionado con los métodos que tenemos de aprender de los datos y el objetivo será intentar hallar una forma eficiente de minimizar tanto el error de la aproximación como la propia penalización.

4. Control de la complejidad. Como hemos dicho antes el control de la complejidad no es algo sencillo y habrá que escoger la mejor manera de medir dicha complejidad. En secciones posteriores veremos medidas de complejidad como la dimensión de Vapnik-Chervonenkis.

Veamos brevemente la distinción que hemos hecho entre la penalización paramétrica y no paramétrica.

Penalización paramétrica

Supongamos que tenemos un conjunto de funciones $f(x, \omega)$ con $\omega \in \Omega$ donde Ω es el conjunto de parámetros de la forma $\omega = (\omega_0, ..., \omega_m)$. Como la aproximación viene definida por el parámetros ω entonces podemos definir también la penalización asociada a dicha selección de parámetros.

Vamos a ver los ejemplos de las penalizaciones más empleadas de este tipo.

- Ridge: $\phi_r(\omega_m) = \sum_{i=0}^{m} \omega_i^2$
- Selección de subconjunto: $\phi_s(\omega_m) = \sum_{i=0}^{m} \chi(\omega_i \neq 0)$
- Bridge: $\phi_p(\omega_m) = \sum_{i=0}^{m} |\omega_i|^p$
- Decaimiento de peso: $\phi_q(\omega_m) = \sum_{i=0}^{m} \frac{(\omega_i/q)^2}{1+(\omega_i/q)^2}$

Penalización no paramétrica

En primer lugar vamos a definir la transformada de Fourier de una función para poder definir el funcional de penalización.

Definición 2.2 (Transformada de Fourier) *Sea f una función integrable Lebesgue, $f \in L(\mathbb{R})$. Se define la transformada de Fourier de f como la función:*

$$\mathcal{F}\{f\} : \xi \to \hat{f}(\xi) := \int_{-\infty}^{\infty} f(x) e^{-2\pi i \xi x} dx$$

Recordemos brevemente las propiedades de la transformada de Fourier.

- La transformada de Fourier es un operador lineal: $\mathcal{F}\{a \cdot f + b \cdot g\} = a\mathcal{F}\{f\} + b \cdot \mathcal{F}\{g\}$
- $\mathcal{F}\{f(at)\}(\xi) = \frac{1}{|a|} \cdot \mathcal{F}\{f\}(\frac{\xi}{a})$
- $\mathcal{F}\{f(t-a)\}(\xi) = e^{-\pi i \xi a} \cdot \mathcal{F}\{f\}(\xi)$
- $\mathcal{F}\{f\}(\xi - a) = \mathcal{F}\{e^{\pi i a t} f(t)\}(\xi)$
- $\mathcal{F}\{f'\}(\xi) = 2\pi i \xi \mathcal{F}\{f\}(\xi)$
- $\mathcal{F}\{f\}'(\xi) = \mathcal{F}\{(-it) \cdot f(t)\}(\xi)$

Habiendo recordado esto podemos definir el funcional de penalización no paramétrica. Este funcional mide la suavidad del ajuste de la función gracias a que se puede medir, mediante la transformada de Fourier, la ondulación de la función. Por tanto el funcional no paramétrico que se propone es:

$$\phi[f] = \int_{\mathbb{R}^d} \frac{|\hat{f}(s)|^2}{\hat{G}(s)} ds$$

Donde $\hat{f}$ indica la transformada de Fourier de la función f y $\frac{1}{\hat{G}}$ es la transformada de Fourier de una función de filtro de paso alto. Es en esta proposición de filtro donde se añade el conocimiento a priori del problema. Por ejemplo pudiera ser interesante en alguna aplicación práctica tener un funcional invariante frente a rotaciones de funciones.

2.3.4. Equilibrio entre el sesgo y la varianza

Este enfoque es muy utilizado en el estudio del error, dividiéndolo en sesgo y varianza para hacer un mejor estudio del mismo y poder enfrentar ambos con varios métodos. Este estudio del caso clásico no es válido (o al

menos no del todo) para problemas no supervisados como es nuestro caso. Vamos a hacer una adaptación de esta teoría para que pueda encajar en nuestro caso de estudio.

Tenemos que tener en cuenta que no conocemos la salida real del sistema en el caso de detección de anomalías, es decir, no sabemos estimar con certeza el sesgo y la varianza y por tanto el error que cometemos. En primer lugar vamos a ver una pequeña adaptación de la notación al caso de detección de anomalías para poder hacer un estudio enfocado.

Vamos a notar por $X_1, ..., X_n$ los datos de test y $\mathcal{D}$ como conjunto de datos de entrenamiento. Además vamos a considerar que existe una función f que nos da la etiqueta real de un dato, esto es, si es o no una anomalía. Por tanto podemos decir que la auténtica etiqueda de un dato es $y_i = f(X_i)$. Además nosotros estaremos usando un modelo ya escogido por nosotros para predecir la etiqueta de un dato de test, esto es $g(X_i, \mathcal{D}) \approx y_i + \beta$ donde β es un cierto error.

Una vez conocida esta notación podemos definir el error medio al cuadrado como:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\{y_i - g(X_i, \mathcal{D})\}^2$$

Y podemos definir también el valor esperado del error medio al cuadrado como:

$$E[MSE] = \frac{1}{n}\sum_{i=1}^{n}E[\{y_i - g(X_i, \mathcal{D}\}^2]$$

Una vez definido el MSE esperado podemos desarrollar un poco el cálculo para poder obtener el error y la varianza que esperamos.

En primer lugar podemos escribirlo como:

$$E[MSE] = \frac{1}{n}\sum_{i=1}^{n}E[\{(y_i - f(X_i)) + (f(X_i) - g(X_i, \mathcal{D}))\}^2]$$

Aquí solo hemos restado y sumado $f(X_i)$, ahora si recordamos que $y_i = f(X_i)$ entonces podemos igualar el primero de los paréntesis a 0 y por tanto nos queda:

$$E[MSE] = \frac{1}{n}\sum_{i=1}^{n} E[\{f(X_i) - g(X_i, \mathcal{D})\}^2]$$

Si seguimos descomponiendo podemos sumar y restar $E[g(X_i, \mathcal{D})]$ con lo que nos quedaría:

$$E[MSE] = \frac{1}{n}\sum_{i=1}^{n} E[\{f(X_i) - E[g(X_i, \mathcal{D})]\}^2]$$

$$+\frac{2}{n}\sum_{i=1}^{n}\{f(X_i) - E[g(X_i, \mathcal{D})]\} \cdot \{E[g(X_i, \mathcal{D})] - E[g(X_i, \mathcal{D})]\}$$

$$+\frac{1}{n}E[\{E[g(X_i, \mathcal{D}) - g(X_i, \mathcal{D})]\}^2]$$

Como es claro, el segundo término da cero por lo que nos queda al final:

$$E[MSE] = \frac{1}{n}\sum_{i=1}^{n} E[\{f(X_i) - E[g(X_i, \mathcal{D})]\}^2] + \frac{1}{n}\sum_{i=1}^{n} E[\{E[g(X_i, \mathcal{D})] - g(X_i, \mathcal{D})\}^2]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\{f(X_i) - E[g(X_i, \mathcal{D})]\}^2 + \frac{1}{n}\sum_{i=1}^{n} E[\{E[g(X_i, \mathcal{D})] - g(X_i, \mathcal{D})\}^2]$$

Si reconocemos cada uno de los términos, en primer lugar el primero de ellos es el sesgo al cuadrado y el segundo la varianza, por lo que finalmente lo que hemos obtenido es:

$$E[MSE] = sesgo^2 + varianza$$

El dilema que se nos plantea es el siguiente: si tomamos modelos con un bajo sesgo en la estimación de los parámetros entonces tendremos una alta varianza y viceversa. Esto significa que no podemos con el conocimiento del que disponemos disminuir tanto el sesgo como la varianza a la vez. Es esta propiedad la que se conoce como la compensación entre sesgo y varianza.

2.4. Teoría estadística del aprendizaje

En esta sección vamos a hacer un repaso por la teoría del aprendizaje, en concreto la teoría desarrollada por Vapnik-Chervonenkis. Esta teoría se basa o tiene como pilares cuatro puntos:

1. Condiciones para la consistencia del principio ERM o minimización del error empírico.
2. Cotas en la capacidad de generalización de las máquinas de aprendizaje.
3. Principios de inferencia sobre muestras finitas.
4. Métodos constructivos para implementar los principios inductivos ya expuestos.

Durante el desarrollo de esta sección haremos un repaso de estos cuatro puntos para dar el broche final a esta sección y poder realizar la primera de las definiciones de anomalía.

2.4.1. Condiciones para la convergencia y consistencia del ERM

En el problema de aprendizaje disponemos de una muestra en la que tenemos los propios datos de entrada y la salida del sistema. Denotemos a estos elementos por $z = (x, y)$ donde x son los datos de entrada e y la salida del sistema. Por tanto la muestra que se nos da es un conjunto $Z_n = \{z_1, ..., z_n\}$. Estos datos están generados como ya sabemos mediante una función de densidad desconocida $p(z)$. Sobre este esquema tenemos una serie de funciones de pérdida y un funcional de pérdida. El objetivo es encontrar dicha función de pérdida $Q(z, \omega)$ que minimice dicho funcional:

$$R(\omega) = \int Q(z, \omega)p(z)dz$$

Por tanto si tenemos la función de pérdida podemos definir el error empírico como:

$$R_{emp}(\omega) = \sum_{i=1}^{n} Q(z_i, \omega)$$

Donde ω son los parámetros escogidos para el modelo.

Para poder estudiar la consistencia del ERM primero debemos definir formalmente dicha propiedad. Denotamos por $R_{emp}(\omega_n^*)$ el valor del error empírico con la función de pérdida $Q(z, \omega_n^*)$ que minimiza el error empírico para el conjunto de entrenamiento Z_n. Denotemos además por $R(\omega_n^*)$ el verdadero valor (desconocido) del error para la función de pérdida. Como se puede ver estos valores dependen del tamaño del conjunto de entrenamiento n, podemos por tanto estudiar cómo se comportan estos errores cuando aumentamos el tamaño del conjunto de entrenamiento. Es aquí donde entra la definición de consistencia del ERM. Decimos que es consistente si la sucesión de errores reales y empíricos convergen en probabilidad al mismo límite $R(\omega_0) = \text{mín}_\omega R(\omega)$. Es decir:

$$R(\omega_n^*) \to R(\omega_0) \textit{ cuando } n \to \infty$$
$$R_{emp}(\omega_n^*) \to R(\omega_0) \textit{ cuando } n \to \infty$$

Para poder asegurar esta propiedad sobre el ERM tenemos el conocido como Teorema Clave de la Teoría del Aprendizaje de Vapnik y Chervonenkis.

Teorema 2.3 (Teorema Clave de la Teoría del Aprendizaje) *Para funciones de pérdida acotadas el principio inductivo de minimización del error empírico es consistente sí y sólo si el error empírico converge uniformemente al valor real del error en el siguiente sentido:*

$$\lim_{n\to\infty} P[\sup_{\omega} |R(\omega) - R_{emp}(\omega)| > \epsilon] = 0 \ , \ \forall \epsilon > 0$$

Cabe recalcar que estas condiciones de consistencia dependen de las propiedades de la clase de funciones elegida. No podemos pretender escoger como clase de aproximación una muy general y seguir manteniendo las condiciones de consistencia del ERM. Aún así el teorema nos está dando condiciones generales para la consistencia del ERM pero son abstractas y no fácilmente aplicables en la práctica. Para ello vamos a estudiar las condiciones de convergencia de ERM que sí serán aplicables en la implementación de algoritmos.

Vamos ahora a particularizar el estudio en el caso de clasificación binaria por ser la materia de estudio que nos ocupa, pues al final tendremos que clasificar instancias en anómalas o no anómalas. Ahora las funciones de pérdida $Q(z, \omega)$ son funciones de pérdida indicadoras. Vamos a notar por $N(Z_n)$ el número de dicotomías que se pueden tener con la clase de funciones

elegidas. Esto es el número de formas de clasificar los datos en las dos clases existentes.

Una vez actualizada nuestra notación podemos definir la entropía aleatoria como $H(Z_n) = \ln N(Z_n)$. Esta cantidad es una variable aleatoria dependiente de los valores de entenamiento Z_n, podemos definir ahora la entropía de Vapnik-Chervonenkis como el valor medio o esperado de la entropía aleatoria:

$$H(n) = E[\ln N(Z_n)]$$

Esta medida es una cuantificación de la diversidad del conjunto de funciones indicadoras que nos pueden separar los datos en ambas clases.

Por último vamos a definir la función de crecimiento que nos va a permitir hacer cotas y llegar a la condición necesaria y suficiente para la convergencia del ERM. Definimos la función de crecimiento como:

$$G(n) = \ln \max_{Z_n} N(Z_n)$$

Donde aquí estamos notando el máximo número de dicotomías sobre todas las posibles muestras existentes de tamaño n. Es más, como el máximo número de formas de dividir un conjunto de tamaño n en dos clases es 2^n entonces podemos afirmar que $G(n) \leq n \ln(2)$.

Por último y para completar la cadena de desigualdades que buscamos vamos a definir la entropía reforzada de Vapnik-Chervonenkis:

$$H_{ann}(n) = \ln(E[N(Z_n)])$$

Haciendo uso de la conocida desigualdad de Jensen

$$\sum_{i=1}^{n} a_i \ln(x_i) \leq \ln(\sum_{i=1}^{n} a_i x_i)$$

podemos ver claramente que $H(n) \leq H_{ann}(n)$. Por tanto obtenemos la cadena de desigualdades:

$$H(n) \leq H_{ann}(n) \leq G(n) \leq n \ln(2)$$

La condición necesaria y suficiente de Vapnik-Chervonenkis para la convergencia del ERM que hallaron fue que:

$$\lim_{n\to\infty} \frac{H(n)}{n} = 0$$

Pero esta condición no asegura una convergencia rápida asintóticamente al error real. Se dice que el ratio de convergencia es rápido asintóticamente en la Teoría de Vapkin-Chervonenkis si:

$$\forall n > n_0 \; P(R(\omega) - R(\omega^*) < \epsilon) = e^{-cn\epsilon^2} \; con \; c > 0$$

Para poder cumplir esta condición se dio la condición suficiente para la convergencia rápida:

$$\lim_{n\to\infty} \frac{H_{ann}(n)}{n} = 0$$

Estas condiciones son dependientes de la distribución de los datos Z_n como es claro al depender de la esperanza de una variable aleatoria dependiente de Z_n. Es por tanto que esta condición no es del todo general. Para solventar esto se tiene la consistencia y convergencia del ERM con la condición necesaria y suficiente de que:

$$\lim_{n\to\infty} \frac{G(n)}{n}$$

Por tanto este estudio nos ha dado las condiciones de convergencia y consistencia del ERM.

2.4.2. Función de crecimiento y dimensión de Vapnik-Chervonenkis

El objetivo que perseguimos es obtener cotas para la capacidad de generalización de las máquinas de aprendizaje. Para dar el primer paso según hemos visto necesitamos una forma de evaluar la función de crecimiento vista en el apartado anterior, cosa que no es sencilla de llevar a la práctica.

Para continuar avanzando en este camino lo primero que vamos a presentar es el concepto de dimensión de Vapnik-Chervonenkis o dimensión VC. Cuando discutimos cómo medir la complejidad de un modelo ya hablamos

de que la dimensión VC podría ser una buena herramienta para este fin, la introducimos a continuación.

Vapnik y Chervonenkis probaron que la función de crecimiento estaba acotada por una función logarítmica en función del tamaño de la muestra. El punto en el que se tiene $n = h$ donde h es un valor fijo se tiene que el crecimiento de la función de crecimiento empieza a ralentizarse, esta es la conocida como dimensión VC. Si h es un número finito entonces tenemos que la función de crecimiento no va a crecer de forma lineal para muestras de tamaño grande y de hecho se tiene la cota:

$$G(n) \leq h(1 + \ln(\frac{n}{h}))$$

La dimensión de Vapnik-Chervonenkis es intrínseca a la elección del conjunto de funciones y además nos da condiciones sobre la convergencia rápida del ERM. Ya hemos visto antes que la cota más grande de la función de crecimiento es:

$$G(n) \leq n \ln(2)$$

Si comparamos las dos cotas en función de n tenemos la siguiente gráfica:

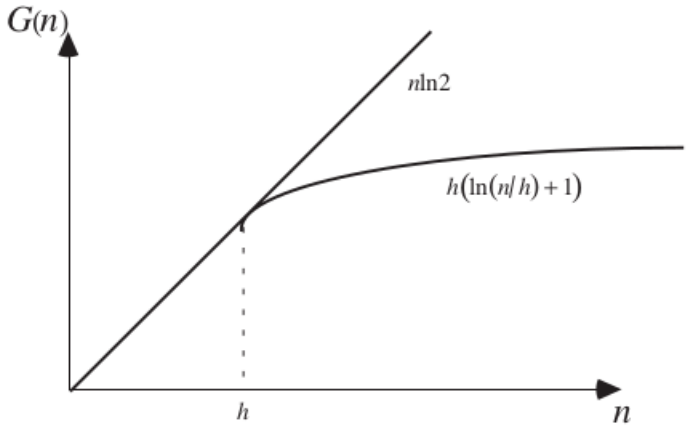

Figura 2.4: Comportamiento de la función de crecimiento [2, p. 107]

Como podemos ver el comportamiento una vez que el tamaño de la muestra alcanza la dimensión VC converge de forma mucho más rápida.

Hasta ahora no hemos definido formalmente la dimensión VC pero hemos dado una característica de la misma que nos garantiza una buena convergencia. Decimos que un conjunto de funciones indicadoras tiene dimensión VC h si existe una muestra de puntos de tamaño h que puede ser dividida pero no existe una muestra de tamaño $h+1$ que cumpla dicha condición. Es decir, podemos decir que la dimensión VC es la máxima dimensión para la que existe una solución óptima a nuestro problema de dividir el conjunto de datos entre datos anómalos y normales.

Veamos esto con un ejemplo gráfico:

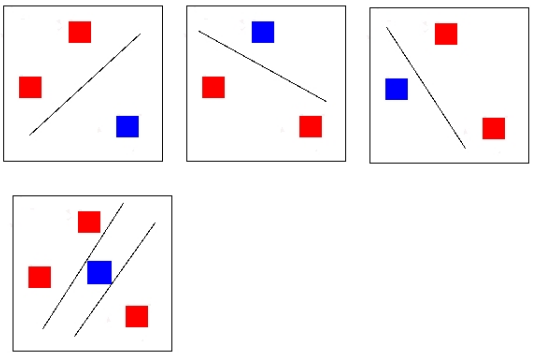

Figura 2.5: Ejemplo de cálculo de dimensión VC Wikimedia

Como podemos ver si estamos considerando funciones lineales para aproximar podemos dividir todas las posibilidades con tamaño de muestra 3, pero no con tamaño de muestra 4 por lo que en este caso concreto la dimensión VC será 3.

Ahora podemos volver a la cadena de desigualdades que hemos visto en la sección anterior y actualizarla con la nueva mejor cota que hemos desarrollado con la dimensión VC:

$$H(n) \leq H_{ann}(n) \leq G(n) \leq h(1 + \ln(\frac{n}{h}))$$

La definición que hemos dado es para funciones indicadoras pero no para funciones reales en general, vamos a generalizar por tanto la definición de la dimensión de Vapnik-Chervonenkis para el caso de funciones reales.

Consideramos como hemos hecho anteriormente funciones de pérdida del tipo $Q(z, \omega)$ pero acotadas superior e inferiormente por constantes:

$$A \leq Q(z, \omega) \leq B$$

Para este caso podemos pensar en una función indicadora que nos diga si $Q(z, \omega)$ está por encima o por debajo de un cierto valor β con $A \leq \beta \leq B$. Podemos por tanto generalizar la dimensión VC para el caso de funciones de pérdida reales como la dimensión VC de este tipo de funciones indicadoras dependientes del parámetro β.

2.4.3. Límites de la generalización

Venimos de discutir las propiedades de convergencia y consistencia del principio inductivo ERM. Ahora bajo este principio vamos a intentar ir un paso mas allá en la generalización e intentar responder a las siguientes dos preguntas:

1. ¿Cómo de cerca están el error real $R(\omega^*)$ y el mínimo error empírico $R_{emp}(\omega^*)$?
2. ¿Cómo de cerca están el error real $R(\omega^*)$ y el mínimo error posible $R(\omega_0) = \text{mín}_\omega\, R(\omega)$?

Estas preguntas las vamos a resolver en el marco que hemos introducido, con todos los conceptos anteriores de la teoría de Vapnik-Chervonenkis en el caso del problema de clasificación binaria que es el que más se ajusta a nuestro problema.

Según Vapnik-Chervonekis se puede acotar el error real cometido por el error empírico con una probabilidad de al menos $1 - \eta$ usando el principio inductivo ERM. La cota que hallaron es la siguiente:

$$R(\omega) \leq R_{emp}\omega) + \frac{\epsilon}{2}\left(1 + \sqrt{1 + \frac{4 \cdot R_{emp}(\omega)}{\epsilon}}\right)$$

donde:

$$\epsilon = a_1 \cdot \frac{h(\ln(\frac{a_2 n}{h}) + 1) - \ln(\frac{\eta}{4})}{n}$$

cuando el conjunto de funciones de pérdida $Q(z, \omega)$ contiene un número infinito de elementos, en caso contrario:

$$\epsilon = 2\frac{\ln(N) - \ln(\eta)}{n}$$

y los valores a_1, a_2 son constantes sobre los que se exigen condiciones.

Para empezar según la demostración de Vapnik, los valores a_1 y a_2 deben estar en los rangos $0 < a_1 \leq 4$ y $0 < a_2 \leq 2$ siendo la pareja de valores $a_1 = 4$ y $a_2 = 2$ la correspondiente al peor de los casos, dando en como resultado el siguiente valor de ϵ:

$$\epsilon = 4\frac{h(\ln(\frac{2n}{h})) - \ln(\frac{\eta}{4})}{n}$$

Con esta desigualdad estamos dando la cota de cómo se comporta el error real con respecto al error empírico. Podemos ver que, en el mejor de los casos el error real y el error empírico se van a diferenciar en al menos $\frac{\epsilon}{2}$.

Además, para resolver la segunda de las preguntas la teoría de Vapnik-Chervonenkis nos da la siguiente cota con probabilidad al menos $1 - 2\eta$:

$$R(\omega_n^*) - \min_{\omega} R(\omega) \leq \sqrt{\frac{-\ln(\eta)}{2n}} + \frac{\epsilon}{2}(1 + \sqrt{1 + \frac{4}{\epsilon}})$$

En este caso podemos ver que la cota es aún mayor que en el caso anterior, teniéndose tanto en esta cota como en la anterior que a mayor nivel de confianza (menor valor de η) mayor es la cota y por tanto menos información tenemos. Es decir, no podemos conocer una buena cota con un nivel de confianza alto. Este hecho no debería de sorprendernos pues seguimos trabajando con un número finito de datos y ya sabemos que no podemos obtener una aproximación todo lo buena que queramos con un número finito de datos de entrenamiento.

Este hecho también se refleja en los valores analíticos que hemos dado. Pensemos en un escenario con $\eta \to 0$, es decir un alto nivel de confianza. Entonces si miramos la expresión de ϵ podemos observar que $-\ln(\frac{\eta}{4}) \to \infty$ y por tanto $\epsilon \to \infty$ con lo que la cota no nos aportaría ninguna información en ninguno de los dos casos.

Por otro lado si lo que crece es el tamaño de la muestra, es decir, $n \to \infty$, estamos aumentando el conocimiento que tenemos sobre el problema y por tanto lo razonable sería que ambas cotas tendieran al valor óptimo. En efecto si observamos el valor de ϵ cuando $n \to \infty$ vemos que tiende a 0 por lo que el error empírico y real están muy cerca y de igual forma el error real y el

mínimo error posible. Por tanto podemos decir que nuestro nivel de certeza depende del tamaño de la muestra. Este hecho fue visto por Vapnik en su teoría y propuso como valor aproximado de la confianza de la desigualdad aquel que lleva asociado el valor:

$$\eta = \text{mín}(\frac{4}{\sqrt{n}}, 1)$$

2.4.4. Principio de minimización del error estructural (SRM)

Hemos visto una construcción en base al principio inductivo ERM y hemos razonado que funciona bien para casos en los que la proporción $\frac{n}{h}$, es decir la proporción del tamaño de la muestra y la dimensión VC, es grande. En este caso quiere decir que tenemos muchos datos comparado con la dimensión VC y por tanto $\epsilon \approx 0$. Por contra cuando tenemos que $\frac{n}{h}$ es pequeño no tenemos mucha información de la cota. Por tanto, al estar el número de datos fijo por el problema, tenemos que buscar un conjunto de funciones para aproximar la salida del sistema que nos den una dimensión VC controlable para hacerla más o menos grande.

El principio inductivo que pretende plasmar esta idea es el principio de minimización del error estructural o SRM. Bajo este principio se le otorga a la clase de funciones de pérdida de una estructura, es decir, tenemos subconjuntos de la forma $S_k = \{Q(z, \omega),\ \omega \in \Omega_k\}$ de forma que:

$$S_1 \subset S_2 \subset ... \subset S_k \subset ...$$

donde cada subconjunto de funciones de pérdida tiene asociada una dimensión VC h_k teniéndose el orden:

$$h_1 \leq h_2 \leq ... \leq h_k \leq ...$$

Al igual que en el Teorema clave de la Teoría del Aprendizaje de Vapnik-Chervonenkis se exigía que las funciones de pérdida estuvieran acotadas en este caso vamos a pedir que las funciones contenidas en cada uno de los S_k o bien estén acotadas o si no que cumplan que:

$$\sup_{\omega \in \Omega_k} \frac{(\int Q^p(z, \omega) dp(z))^{\frac{1}{p}}}{\int Q(z, \omega) dp(z)} \leq \tau_k,\ p > 2$$

para alguna pareja (p, τ_k).

En cuanto a la definición del SRM hay dos estrategias prácticas que se llevan a cabo para su implementación que son:

1. Mantener la dimensión VC fija y minimizar el error empírico.
2. Mantener el error empírico constante y pequeño y minimizar la dimensión VC.

Estas implementaciones realmente quedan muy libres en la práctica y se proponen por tanto diferentes estructuras de minimización del error empírico y la dimensión VC que se saben que funcionan bien.

Por tanto este principio no se basa meramente en el buen ajuste de los datos, si no que además pretende hacer una minimización de la complejidad del modelo propuesto.

2.4.5. Aproximaciones de la dimensión VC

Como hemos estado viendo las cotas que hemos expuesto y desarrollado dependen en mayor o menor medida de la dimensión VC. Como es lógico este valor no es fácil de calcular, y de hecho sólo se sabe para unos cuantos conjuntos de funciones de aproximación. Vapnik propuso una método para poder estimar este valor y así poder obtener unas cotas aproximadas.

El procedimiento propuesto por Vapnik consiste en, dadas dos muestras Z_n^1, Z_n^2 de tamaño n de pares $z_i = (x, y)$ de datos de entrada y salida del sistema vamos a medir el error empírico con nuestro modelo que cometemos observando la máxima desviación de los ratios de error de estas dos muestras independientes, es decir:

$$\xi(n) = \max_{\omega}(|Error(Z_n^1) - Error(Z_n^2|)$$

donde $Error(Z_n^i)$ es la tasa de error empírico cometido por el modelo. De acuerdo con la teoría desarrollada por Vapnik-Chervonenkis tenemos que $\xi(n)$ está acotada:

$$\xi(n) \leq \Phi(\frac{n}{h})$$

donde h es la dimensión VC y:

$$\Phi(\tau) = \begin{cases} 1 & si\ \tau < 0.5 \\ a\frac{\ln(2\tau)+1}{\tau-k}\left(\sqrt{1+\frac{b(\tau-k)}{\ln(2\tau)+1}}+1\right) & en\ otro\ caso \end{cases}$$

donde $\tau = \frac{n}{h}$ y las constantes $a = 0.16$ y $b = 1.2$ son constantes estimadas empíricamente por Vapnik y $k = 0.14928$ tomada así para que $\Phi(0.5) = 1$ de forma que la cota sea muy ajustada.

Por tanto describió con esto el siguiente esquema de obtención de la aproximación de la dimensión VC:

1. Generamos una muestra de tamaño $2n$ etiquetada z_{2n}
2. Dividimos la muestra en dos del mismo tamaño Z_n^1 y Z_n^2
3. Invertir las etiquetas de Z_n^2
4. Mezclar los dos conjuntos de nuevo y entrenar el modelo
5. Separar el conjunto en dos de nuevo e invertir las etiquetas del segundo, volver a mezclarlos y entrenar de nuevo el modelo
6. Medir la diferencia de los errores $\xi(n) = |Error(Z_n^1) - Error(Z_n^2)|$

Como hemos dicho antes la desigualdad $\xi(n) \leq \Phi(\frac{n}{h})$ es muy ajustada y por tanto, podemos obtener la aproximación de h como:

$$h^* = arg\min_h[\xi(n) - \Phi(\frac{n}{h})]$$

Es decir, el valor que haga dicha diferencia menor, es decir que más acerque los valores $\xi(n)$ y $\Phi(\frac{n}{h})$.

2.4.6. Perspectiva

Tras este desarrollo teórico hemos dado un marco sobre el cuál podremos cimentar el resto del trabajo y modelos empleados. Con este capítulo hemos hecho un repaso por toda la teoría básica de Machine Learning y la teoría desarrollada por Vapnik y Chervonenkis.

El siguiente paso que debemos dar es introducir la definición clásica de anomalía basada en distancias y dar algunas reflexiones sobre ella para poder proseguir en el estudio.

Capítulo 3

Concepto de anomalía

3.1. Contextualización

Ya hemos discutido previamente una idea intuitiva del concepto de anomalía. Un dato decimos que es anómalo cuando se distancia del resto de los datos lo suficiente como para no tener características comunes con el resto.

Este hecho puede ser por distintos motivos. Puede que la anomalía venga del hecho de que se está produciendo un evento en nuestros experimentos que no sea nada frecuente. Por ejemplo podemos estar midiendo datos meteorológicos y que en un momento dado se den una serie de fenómenos que no sea frecuente ver juntos, o incluso que no se hayan visto nunca ocurrir simultáneamente. Otra forma de tener una anomalía en nuestro conjunto de datos pudiera ser errores de medición. Por ejemplo si seguimos con este símil de los datos meteorológicos imaginemos que nuestra estación dispone de un termómetro. Este sensor se ha roto y empieza a marcar datos superiores a 100 C°, claramente son datos muy desviados de las temperaturas normales con lo que no tendrían relación con el resto y presentaría una desviación muy importante con respecto al resto de los datos.

3.2. Criterios

Esta idea intuitiva que estamos dando de anomalía no refleja todos los posibles escenarios. Los ejemplos que estamos dando suponen una desviación muy grande de los datos normales, tanto que no se pueden comparar con el resto porque difieren mucho numéricamente. Vamos a plantear un escenario para dar una mejor forma al concepto de anomalía. Pensemos en una serie

de datos muy agrupados en dos clústers por ejemplo:

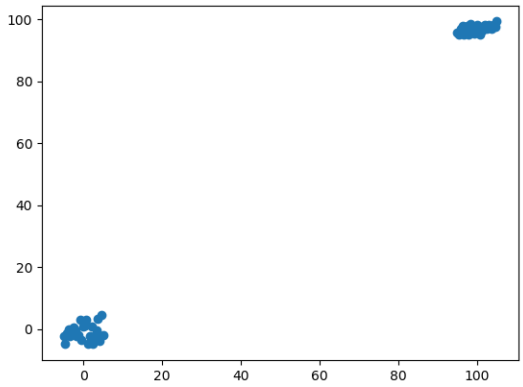

Figura 3.1: Clusters alejados

Como podemos comprobar que tenemos dos clústers no sólo alejados entre sí, si no con los elementos muy concentrados para poner un caso extremo. Ahora no vamos a proponer un valor que se aleje de los dos clústers, si no uno que esté a medio camino entre los dos:

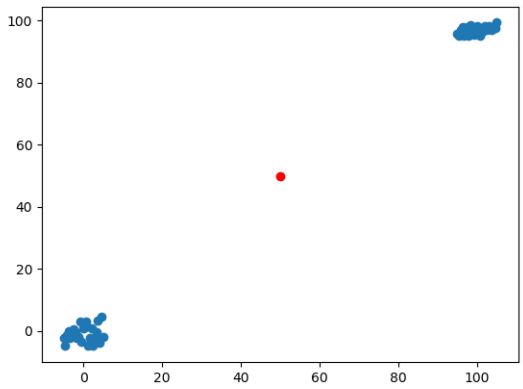

Figura 3.2: Clusters alejados con una anomalía en rojo

Si los datos del clúster de abajo a la izquierda fueran datos de temperatura con valores entorno a 0 y los de arriba a la derecha fueran de datos de temperatura entorno a 100 grados nuestro datos anómalo tendría una temperatura de unos 50 grados. Esta temperatura no se aleja radicalmente

de los valores normales, es decir, no son -1000 grados ni 1000 grados. Aún así estamos describiendo una situación anómala.

No podemos dar una definición formal o que podamos decir que abarca todos los casos para definir lo que es una anomalía, aún así vamos a intentar dar dos puntos de vista: uno basado en distancias y otro en probabilidades.

El criterio más usado en la definición o detección de anomalías es el llamado "Tukey's Fences". Para introducirlo vamos a ver su definición en una única dimensión para luego extender el concepto. Pensemos en un conjunto de datos 1-D. Sobre sus valores podemos calcular los cuartiles Q_1, Q_2 y Q_3. Un valor anómalo es aquel que no cae dentro del intervalo $[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)]$ donde k es una constante. El valor propuesto para k por Tukey fue de $k = 1.5$ aunque algunos autores más restrictivos proponen $k = 3$.

Este criterio puede ser extendido al caso de mayor dimensionalidad si realizamos este mismo test sobre todos los valores de todas las características y comprobar si alguno o todos se salen del rango en función de cómo de restrictivo queremos que sea el criterio.

Esta extensión es muy vaga, por lo que se propone un criterio un poco más fijado. Imaginemos los datos agrupados por clústers, entonces podemos fijar un centroide de dicho cluster. Sobre cada cluster podemos medir cuál es la mayor distancia intercluster de los datos al centroide. Podemos extender el criterio de Tukey diciendo que un dato anómalo es aquel que se distancia más de 1.5 veces de la mayor distancia intercluster al centroide.

Esta generalización ya si abarca el ejemplo que hemos propuesto. Al estar muy apiñados los datos entorno al centroide la mayor distancia intercluster es muy pequeña, de hecho en el ejemplo construido es menor que 5. Por tanto el dato $(50, 50)$ está alejado más de $1.5 \cdot 5 = 7.5$ unidades del centroide y por tanto lo podemos considerar una anomalía.

3.3. Qué hacer con las anomalías

Estamos estudiando cómo podemos detectar anomalías pero una vez que las hayamos detectado en nuestros conjuntos de datos en un problema real, ¿qué debemos hacer con ellas? Este problema es algo muy general, puede que estemos seguros en nuestro caso de que las anomalías se han debido a un problema de medición porque nuestros instrumentos estaban rotos y por tanto deberíamos descartarlos. Puede que sean datos reales pero estén tan separados del resto que debamos estudiarlos de forma separada. Vamos a

dar unas cuantas alternativas a lo que podemos hacer una vez que hemos detectados las anomalías dentro de nuestro conjunto de datos:

- Dejarlos: puede que nuestro conjunto de datos tenga un número de datos muy elevado y por tanto aparezcan en él anomalías. Estas anomalías no deben ser eliminadas, es más debemos escoger modelos que aprendan o utilicen los datos teniendo en cuenta estas anomalías y siendo robustos ante su aparición.
- Exclusión: una opción es eliminar directamente las anomalías. Esto en general no está justificado y de hecho no se recomienda pues perdemos riqueza del conjunto de datos al eliminar instancias del mismo. Aún así, si decidimos eliminar los datos tenemos dos formas de hacerlo. Podemos eliminarlos directamente y prescindir de esos datos o podemos sustituirlos por datos cercanos que no sean anómalos.
- Estudiarlos por separado: puede que tengamos un número suficientemente elevado de las mismas y que enseñen algunos patrones o tengan explicación en nuestro ejemplo del mundo real. En este caso quizás deberíamos considerarlas y estudiarlas a parte para darles un sentido y emplear el conocimiento que les subyace.

Capítulo 4

Introducción de Estadística Multivariante

Vamos a dar otra definición de anomalía que no coincide con la que hemos visto basada en distancias, pero antes de dar esa definición debemos hacer un breve repaso de estadística multivariante y probabilidad para poder comprender y enmarcar dicha definición.

4.1. Introducción

En primer lugar vamos a describir conceptos básicos sobre los que poder construir los conceptos que necesitamos para la definición de anomalía basada en probabilidades.

En primer lugar vamos a definir el concepto de variable aleatoria.

Definición 4.1 *Una variable aleatoria es una función $X : \Omega \to E$ que parte de un espacio de probabilidad $(\Omega, \mathcal{F}, \mathcal{P})$ y llega a un espacio medible $(E, \mathcal{B})$, donde X además es una función medible.*

Normalmente ya sabemos que $E \subseteq \mathbb{R}$ y además cabe recordar que $\mathcal{F}$ es una σ-álgebra. Además cabe recordar la definición de función medible:

Definición 4.2 *Decimos que una función $X : (\Omega, \mathcal{F}, \mathcal{P}) \to (E, \mathcal{B})$ es medible si $X^{-1}(B) \subset \mathcal{F}$, $\forall B \in \mathcal{B}$.*

Esta definición puede extenderse al caso vectorial, introduciendo con esto la noción de vector aleatorio:

Definición 4.3 *Un vector aleatorio* $\underline{X} = (X_1, ..., X_p)$ *es una aplicación medible* $\underline{X} : (\Omega, \mathcal{F}, \mathcal{P}) \to (E, \mathcal{B}^p)$ *donde* $E \subseteq \mathbb{R}^p$.

Se puede demostrar además la caracterización:

Proposición 4.1 *Un vector* $\underline{X} = (X_1, ..., X_p)$ *es un vector aleatorio si y sólo si* $X_i : (\Omega, \mathcal{F}, \mathcal{P}) \to (\mathbb{R}, \mathcal{B})$ *es una función medible.*

Con este vector aleatorio podemos estudiar o definir la distribución de probabilidad del mismo sobre $(\mathbb{R}^p, \mathcal{B}^p)$ $P_{\underline{X}}$ como:

$$P_{\underline{X}}[B] := P[\underline{X}^{-1}(B)] \ \forall B \in \mathcal{B}$$

con lo que el espacio $(\mathbb{R}^p, \mathcal{B}^p, P_{\underline{X}})$ es un espacio de probabilidad o probabilístico.

Sobre los conocimientos de la definición de la función de distribución univariante podemos hacer una definición análoga para el caso multivariante.

Definición 4.4 *Se define la función de distribución asociada a la probabilidad inducida como:*

$$F_{\underline{X}}(\underline{x}) = P_{\underline{X}}[X_1 \leq x_1, ..., X_p \leq x_p] \ , \ \forall \underline{x} = (x_1, ..., x_p) \in \mathbb{R}^p$$

De igual forma podemos caracterizar la función de densidad como aquella $f_{\underline{X}}$ que, de existir, cumple que:

$$F_{\underline{X}}(\underline{x}) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} ... \int_{-\infty}^{x_p} f_{\underline{X}}(u_1, ..., u_p) du_1 ... du_p$$

Otra forma de determinar de forma única la distribución de un vector aleatorio es mediante la función característica, lo que nos va a dar además una caracterización de la independencia que introduciremos en siguiente lugar.

Definición 4.5 *Dado un vector aleatorio* $X = (X_1, ..., X_p)$ *se define la función característica como* $\Phi_{\underline{X}}(\underline{t}) = E[e^{i\underline{t}X}]$ *con* $\underline{t} = (t_1, ..., t_p) \in \mathbb{R}^p$ *donde la función* $E[\cdot]$ *denota la esperanza, por lo que:*

$$\Phi_{\underline{X}}(\underline{t}) = \int_{\mathbb{R}^p} e^{i\underline{t}X} P_{\underline{X}}(d\underline{x})$$

Con esto ya podemos introducir el concepto de independencia en varias variables.

4.1.1. Independencia

Definición 4.6 *Dados dos vectores aleatorios* $\underline{X} = (X_1, ..., X_p)$, $\underline{Y} = (Y_1, ..., Y_p)$ *se dice que son independientes si:*

$$F_{\underline{X},\underline{Y}}(\underline{x}, \underline{y}) = F_{\underline{X}}(\underline{x}) \cdot F_{\underline{Y}}(\underline{y})$$

Podemos también definir la independencia entre las variables de un vector aleatorio como:

Definición 4.7 $X = (X_1, ..., X_p)$ *se dice que está compuesto de variables independientes si* $\forall B = B_1 \times ... \times B_p$ *con* $B_i \in \mathcal{B}$ *se tiene que:*

$$P_{\underline{X}}(B) = P_{X_1}[B_1] \cdot ... \cdot P_{X_p}[B_p]$$

En cuanto a la independencia de sucesos podemos dar dos definiciones de independencia:

Definición 4.8 *Decimos que los eventos* $B = (B_1, ..., B_p)$ *son independientes dos a dos si para todos* $m \neq k$ *se tiene que* $P(B_m \bigcap B_k) = P(B_m)P(B_k)$

Definición 4.9 *Se dice que los eventos* $B = (B_1, ..., B_p)$ *son independientes mutuamente si para todo* $k \leq p$ *se tiene que* $P(\bigcap_{i=1}^{k} B_i) = \prod_{i=1}^{k}(B_i)$

En cuanto a la definición de independencia entre las variables aleatorias que definen un vector aleatorio podemos dar dos caracterizaciones basadas en la función característica.

Proposición 4.2 *Si las componentes del vector aleatorio $X = (X_1, ..., X_p)$ son independientes entonces:*

$$\Phi_{\underline{X}}(\underline{t}) = E[e^{it\underline{X}}] = \prod_{j=1}^{p} E[e^{it_j X_j}]$$

Proposición 4.3 *Si las componentes del vector aleatorio $X = (X_1, ..., X_p)$ son independientes entonces la función característica de la variable $Y = \sum_{j=1}^{p} X_j$ es:*

$$\Phi_Y(t) = E[e^{itY}] = E[e^{it\sum_{j=1}^{p} X_j}] = \prod_{j=1}^{p} \Phi_{X_j}(t)$$

4.1.2. Probabilidad y esperanza condicionada

En esta sección vamos a describir la probabilidad y esperanza condicionada de una variable aleatoria y no de un vector aleatorio. Este hecho es sencillo de deducir, pues como hemos introducido previamente la distribución de probabilidad de un vector aleatorio viene determinada por una distribución de probabilidad de una variable aleatoria. Por tanto el estudio de la probabilidad y esperanza condicionada en el caso univariante se hace válido para el caso multivariante.

En primer lugar debemos introducir el concepto de probabilidad condicionada tal y cómo la conocemos hasta ahora de Bayes. Partimos de un espacio de probabilidad $(\Omega, \mathcal{A}, \mathcal{P})$.

Definición 4.10 *Definimos la probabilidad condicionada a un suceso $B \in \mathcal{A}$ con $P(B) > 0$ como:*

$$P(\cdot|B) : \mathcal{A} \to [0, 1], \quad P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Esta es una función de probabilidad, por lo que nos lleva a pensar en el espacio de probabilidad que genera, es más podemos pensar en el espacio de probabilidad en el que la probabilidad condicionada no se anula, es decir:

$$\mathcal{A}_B = \{C = A \cap B,\ A \in \mathcal{A}\}$$

Por tanto solemos considerar como espacio de probabilidad condicionada al espacio $(B, \mathcal{A}_B, P(\cdot|B))$.

Partiendo de este espacio de probabilidad podemos considerar una variable aleatoria $X : (\Omega, \mathcal{A}, \mathcal{P}(\cdot|B)) \to (\mathbb{R}, \mathcal{B})$.

Definición 4.11 *Definimos la esperanza de esta variable aleatoria condicionada a B como:*

$$E[X|B] = \int_\Omega X dP(\cdot|B) = \int_\Omega X dP(\cdot|B) = \frac{1}{P(B)} \int_B X dP = \frac{E[X1_B]}{P(B)}$$

Donde 1_B representa la función indicadora del conjunto B.

No sólo podemos estudiar la probabilidad y esperanzas condicionadas a un evento, si no que también las podemos estudiar condicionadas a una σ-álgebra. En este terreno vamos a distinguir dos posibilidades: condicionamiento a una σ-álgebra generada por una partición numerable de sucesos de probabilidad no nula y condicionamiento a una σ-álgebra arbitraria.

Definición 4.12 *Definimos la esperanza condicionada a una σ-álgebra $\mathcal{A}$ generada por $\{B_n\} \subset \mathcal{A}$ con $B_i \cap B_j = \phi,\ i \neq j$, $\bigcup_{n=1}^{\infty} B_n = \Omega$ y $P(B_i) > 0,\ \forall i$. Siendo la $\mathcal{U} = \sigma(\{B_n\})$ la σ-álgebra generada por $\{B_n\}$. Con este marco, definimos la esperanza de una variable aleatoria $X : (\Omega, \mathcal{A}, P) \to (\mathbb{R}, \mathcal{B})$ condicionada a la σ-álgebra $\mathcal{U}$ como:*

$$E[X|\mathcal{U}](\omega) = \sum_{n=1}^{\infty} E[X|B_n]1_{B_n}(\omega)$$

Propiedades 4.1

1. *$E[X|\mathcal{U}] : (\Omega, \mathcal{U}) \to (\mathbb{R}, \mathcal{B})$ es $\mathcal{U}$-medible.*
2. *$E[E[X|\mathcal{U}]] = \sum_{n=1}^{\infty} E[X|B_n]P(B_n) = \sum_{n=1}^{\infty} E[X1_{B_n}] = E[X]$*

De igual forma podemos definir la probabilidad condicionada a una σ-álgebra generada por una partición numerable de sucesos no nulos.

Definición 4.13 *Definimos la probabilidad de un suceso $A \in \mathcal{A}$ condicionada a la σ-álgebra $\mathcal{U}$ como:*

$$P(A|\mathcal{U}) = E[1_A|\mathcal{U}] = \sum_{n=1}^{\infty} E[1_A|B_n]1_{B_n} = \sum_{n=1}^{\infty} P(A|B_n)1_{B_n}$$

casi seguramente.

Podemos también dar unas propiedades inmediatas de la probabilidad condicionada tomando como base las de la esperanza.

Propiedades 4.2 *1.* $P(A|\mathcal{U})$ *es* $\mathcal{U}$*-medible.*

2. $E[P(A|\mathcal{U})] = P(A)$

Una vez visto esto podemos hacer una definición con una σ-álgebra arbitraria. Cabe decir que en este caso no vamos a poder dar una definición constructiva y fácil de calcular como sí hemos hecho en el caso particular anterior. Lo que sí vamos a tener con esta definición más general es el mantenimiento de las propiedades que hemos visto en primera instancia tanto de la probabilidad como de la esperanza condicionada. Sobra decir además que esta definición coincide con la anterior en el caso particular de una σ-álgebra generada por una partición numerable de sucesos no nulos.

Definición 4.14 *Definimos la esperanza de una variable aleatoria* X *en el marco dado condicionada a una* σ*-álgebra* $\mathcal{U} \subset \mathcal{A}$ *como la única función* $\mathcal{U}$*-medible tal que:*

$$\forall u \in \mathcal{U} \ \int_{\mathcal{U}} E[X|\mathcal{U}]P_{\mathcal{U}} = \int_{\mathcal{U}} X dP$$

casi seguramente $P_{\mathcal{U}}$*. Donde* $\forall u \in \mathcal{U} \ P_{\mathcal{U}}(u) = P(u)$*.*

Igualmente podemos dar una definición de la probabilidad condicionada a una σ-álgebra arbitraria tomando como base la definición de esperanza condicionada.

Definición 4.15 *Definimos la probabilidad de* $A \in \mathcal{A}$ *condicionada a la* σ*-álgebra* $\mathcal{U}$ *como:*

$$P(A|\mathcal{U}) = E[1_A|\mathcal{U}]$$

casi seguramente $P_{\mathcal{U}}$*.*

Por último antes de dar unas propiedades que nos den un poco más de conocimiento y herramientas de trabajo vamos a ver el concepto de probabilidad y esperanza condicionada a una variable aleatoria y no a un suceso o una σ-álgebra como hemos visto previamente.

Partimos igualmente del marco $(\Omega, \mathcal{A}, P)$ con dos variables aleatorias X, Y.

Definición 4.16 *Definimos la σ-álgebra generada por la variable aleatoria Y como la menor σ-álgebra que hace medible a la variable aleatoria Y y la notaremos como $\sigma(Y)$.*

Ahora si podemos definir la esperanza de una variable aleatoria condicionada a otra.

Definición 4.17 *Definimos la esperanza de la variable aleatoria X condicionada a la variable aleatoria Y como:*

$$E[X|Y] = E[X|\sigma(Y)]$$

Como anotación cabe decir que esta esperanza condicionada es una función dependiente de la variable aleatoria Y, es decir podemos expresarla como:

$$g(y) = E[X|Y = y]$$

Ahora que tenemos la definición de la esperanza condicionada a una variable aleatoria podemos usar el concepto como hemos hecho anteriormente para definir la probabilidad de un suceso condicionado a una variable aleatoria.

Definición 4.18 *Para todo $A \in \mathcal{A}$ definimos la probabilidad de A condicionada a la variable aleatoria Y como:*

$$P(A|Y) = E[1_A|\sigma(Y)]$$

casi seguramente $P_{\sigma(Y)}$

Ahora estamos en condiciones de dar una propiedades elementales y de suavizamiento que nos van a dar herramientas con las esperanzas condicionadas. En este punto ya hemos visto que, al haber hecho las definiciones de esperanza y probabilidades usándolas indistintamente las propiedades que vamos a dar para la esperanza se pueden emplear para las probabilidades utilizando sus definiciones que impliquen el uso de esperanzas.

Sobre estas propiedades vamos a realizar algunas de las demostraciones de las propiedades elementales y de las de suavizamiento que vamos a dar para poner de relieve cómo podemos hacer uso de la probabilidad y esperanza condicionada.

Propiedades 4.3 (Propiedades elementales) *Partimos de un espacio de probabilidad $(\Omega, \mathcal{A}, P)$, $\mathcal{U}$ una σ-álgebra contenida en $\mathcal{A}$ y X, Y variables aleatorias integrables.*

1. $E[cte|\mathcal{U}] = cte$ *casi seguramente* $P_{\mathcal{U}}$
2. *Sean* $a, b \in \mathbb{R}$ $E[aX + bY|\mathcal{U}] = aE[X|\mathcal{U}] + bE[Y|\mathcal{U}]$ *casi seguramente* $P_{\mathcal{U}}$*, es decir, la esperanza condicionada cumple la propiedad de linealidad.*
3. $X \geq Y$ *casi seguramente* $P \Rightarrow E[X|\mathcal{U}] \geq E[Y|\mathcal{U}]$ *casi seguramente* $P_{\mathcal{U}}$*.*
4. $|E[X|\mathcal{U}]| \leq E[|X||\mathcal{U}]$

Demostración 4.1 *Vamos a demostrar la propiedad 1 para ver como trabajar con las igualdades casi seguras.*

1. *Como la igualdad es casi seguramente podemos aplicar integrales en la misma con lo que obtenemos lo siguiente:*

   $$\forall u \in \mathcal{U} \int_u E[cte|\mathcal{U}]dP_{\mathcal{U}} = \int_u ctedP = cteP(u) = cteP_{\mathcal{U}}(u) = \int_u ctedP_{\mathcal{U}}$$

   Como la igualdad es con integrales, podemos decir por tanto que $E[cte|\mathcal{U}] = cte$ *casi seguramente* $P_{\mathcal{U}}$*.*

Propiedades 4.4 (Propiedades de suavizamiento) *Partimos del marco del espacio probabilístico $(\Omega, \mathcal{A}, P)$ con una σ-álgebra $\mathcal{U} \subset \mathcal{A}$.*

1. *Si X es una variable aleatoria integrable y $\mathcal{U}$-medible entonces se tiene que $E[X|\mathcal{U}] = X$ casi seguramente $P_{\mathcal{U}}$*
2. *Sean X, Y variables aleatorias con X $\mathcal{U}$-medible, Y integrable y XY integrable, entonces se tiene que $E[XY|\mathcal{U}] = XE[Y|\mathcal{U}]$ casi seguramente $P_{\mathcal{U}}$.*
3. *Se dice que X es independiente de $\mathcal{U}$ si X y $1_{\mathcal{U}}$ son independientes. Si X es independiente de $\mathcal{U}$ entonces $E[X|\mathcal{U}] = E[X]$ casi seguramente $P_{\mathcal{U}}$.*
4. *Sean $\mathcal{U}_1 \subset \mathcal{U}_2 \subset \mathcal{A}$ y X una variable aleatoria integrable, entonces:*

   $$E[X|\mathcal{U}_1] = E[E[X|\mathcal{U}_1]|\mathcal{U}_2] = E[E[X|\mathcal{U}_2]|\mathcal{U}_1]$$

   casi seguramente $P_{\mathcal{U}}$.

Vamos a hacer la demostración de las 4 propiedades para dar así una pincelada de cómo aplicar los conceptos vistos hasta ahora.

Demostración 4.2 *Demostremos las propiedades de suavizamiento:*

4. *Sabemos que* $E[X|\mathcal{U}_1] = Z$ *es* $\mathcal{U}_1$*-medible y por tanto es* $\mathcal{U}_2$*-medible, por lo que* $E[Z|\mathcal{U}_2] = Z$ *casi seguramente* $P_{\mathcal{U}_2}$*.*

   Vamos a utilizar ahora el hecho de que las igualdades son casi seguramente y por tanto vamos a ver si aplicando integrales en ambos lados de la igualdad obtenemos el mismo resultado y confirmamos la igualdad.

   $\forall u \in \mathcal{U}_1 \subset \mathcal{U}_2$ *tenemos* $\int_u E[E[X|\mathcal{U}_1]|\mathcal{U}_2]dP_{\mathcal{U}_2} = \int_u E[X|\mathcal{U}_1]dP_{\mathcal{U}_1} = \int_u XdP$

   Veamos ahora desarrillando el otro término.

   $\int_u E[E[X|\mathcal{U}_2]|\mathcal{U}_1]dP_{\mathcal{U}_1} = \int_u E[X|\mathcal{U}_2]dP_{\mathcal{U}_2} = \int_u XdP$

   Al haber llegado a la misma igualdad en integrales tenemos por tanto la igualdad casi seguramente que buscábamos.

1. *Como* X *es* $\mathcal{U}$*-medible entonces tenemos que* $\forall u \in \mathcal{U}\ \int_u E[X|\mathcal{U}]dP_{\mathcal{U}} = \int_u XdP = \int_u XdP_{\mathcal{U}}$ *pues al ser* $\mathcal{U}$*-medible tenemos que* $E[X] = \int_\Omega XdP = \int_\Omega XdP_{\mathcal{U}}$*.*

3. $\forall u \in \mathcal{U}\ \int_u E[X|\mathcal{U}]dP_{\mathcal{U}} = \int_u XdP = \int_\Omega 1_u XdP = E[1_u X] =$
   $= E[1_u]E[X] = P(u)E[X] = P_{\mathcal{U}}(u)E[X] = \int_u E[X]dP_{\mathcal{U}}$

Ya hemos dado las definiciones y propiedades de probabilidad y esperanza condicionadas, para finalizar vamos a ver algunas desigualdades famosas que utilizaremos y sus demostraciones.

4.1.3. Desigualdades y fórmulas famosas

Teorema 4.1 (Desigualdad de Markov) *Sea* X *una variable aleatoria que toma valores no negativos. Entonces para cualquier constante* α *satisfaciendo* $E[X] < \alpha$ *se cumple que:*

$$P(X > \alpha) \leq \frac{E[X]}{\alpha}$$

Demostración 4.3 *Denotemos como* $f_X(x)$ *la función de densidad de la variable aleatoria* X*. Entonces tenemos:*

$$E[X] = \int_x x f_X(x)dx = \int_{0 \leq x \leq \alpha} x f_X(x)dx + \int_{x>\alpha} x f_X(x)dx$$

$$\geq \int_{x>\alpha} x f_X(x)dx \geq \int_{x>\alpha} \alpha f_X(x)dx$$

La primera de las desigualdades se sigue de la no negatividad de X y la segunda se sigue de que la integral está definida sobre los puntos en los que $x > \alpha$, de hecho:

$$\int_{x>\alpha} \alpha f_X(x)dx = \alpha P(X > \alpha)$$

Con lo que tenemos finalmente que:

$$E[X] \geq \alpha P(X > \alpha) \Leftrightarrow P(X > \alpha) \leq \frac{E[X]}{\alpha}$$

■

Teorema 4.2 (Desigualdad de Chebychev) *Sea X una variable aleatoria arbitraria. Entonces para cualquier constante α se tiene que:*

$$P(|X - E[X]| > \alpha) \leq \frac{Var[X]}{\alpha^2 d}$$

Demostración 4.4 *Sabemos que la desigualdad $|X - E[X]| > \alpha$ es cierta si y sólo si $|X - E[X]|^2 > \alpha^2$*

Vamos a definir la variable aleatoria $Y = (X - E[X])^2$ es es no negativa. Con esta definición se tiene que $E[Y] = Var[X]$ por la propia definición de la variable aleatoria Y.

Entonces la parte izquierda de la desigualdad del teorema se puede expresar como $P(|X - E[X]| > \alpha) = P(Y > \alpha^2)$. Aplicando aquí la desigualdad de Markov obtenemos que:

$$P(Y > \alpha^2) \leq \frac{E[Y]}{\alpha^2} = \frac{Var[X]}{\alpha^2}$$

■

Teorema 4.3 (Cota inferior de Chernoff) *Sea X una variable aleatoria que se puede expresar como la suma de N variables aleatorias independiente de Bernoulli, cada una tomando el valor 1 con probabilidad p_i.*

$$X = \sum_{i=1}^{N} X_i$$

Entonces para todo $\delta \in (0, 1)$ tenemos que:

$$P(X < (1 - \delta)E[X]) < e^{-E[X]\delta^2/2}$$

Teorema 4.4 (Cota superior de Chernoff) *Sea X una variable aleatoria que se puede expresar como la suma de N variables aleatorias independiente de Bernoulli, cada una tomando el valor 1 con probabilidad p_i.*

$$X = \sum_{i=1}^{N} X_i$$

Entonces para todo $\delta \in (0, 2 \cdot e - 1)$ tenemos que:

$$P(X > (1 + \delta)E[X]) < e^{-E[X]\delta^2/2}$$

Ambas cotas disponen de una demostración que no es constructiva, por lo que no es relevante su demostración para el estudio. Como ejemplo de una demostración de un estilo similar haremos la demostración de la siguiente desigualdad.

Teorema 4.5 (Desigualdad de Hoeffding) *Sea X una variable aleatoria que se puede expresar como suma de N variables aleatorias independientes acotadas en intervalos $[l_i, u_i]$.*

$$X = \sum_{i=1}^{N} X_i$$

Entonces para todo $\theta > 0$ se tienen las cotas:

$$P(X - E[X] > \theta) \leq e^{-\frac{2\theta^2}{\Sigma_{i=1}^{N}(u_i - l_i)^2}}$$

$$P(E[X] - X > \theta) \leq e^{-\frac{2\theta^2}{\sum_{i=1}^{N}(u_i - l_i)^2}}$$

Demostración 4.5 *Sólo haremos la demostración de la primera desigualdad de forma resumida y sin entrar en los detalles más complejos que se alejan del interés del estudio.*

En primer lugar debemos probar que para todo $t \geq 0$ *se cumple la desigualdad:*

$$P(X - E[X] > \theta) = P(e^{t(X-E[X])} > e^{t\theta})$$

Usando la desigualdad de Markov podemos probar que $P(e^{t(X-E[X])} > e^{t\theta})$ *es como mucho* $E[e^{(X-E[X])}]e^{-t\theta}$.

Además al ser variables aleatorias independientes las que componen la variable aleatoria X *podemos descomponer el término teniendo la desigualdad:*

$$P(X - E[X] > \theta) \leq e^{-t\theta} \prod_i E[e^{t(X_i - E[X_i])}]$$

Cada uno de los términos de este producto se puede probar que vale como mucho $e^{t^2(u_i - l_i)/8}$ *usando argumentos de convexidad y el Teorema de Taylor.*

Por tanto se cumple:

$$P(X - E[X] > \theta) \leq e^{-t\theta} \prod_i e^{t^2(u_i - l_i)^2/8}$$

Nos interesa hallar el valor de $t = t^*$ *que ajusta la desigualdad. Puede demostrarse que ese valor es:*

$$t^* = \frac{4\theta}{\sum_{i=1}^{N}(u_i - l_i)^2}$$

Sustituyendo en la desigualdad con este valor de t *tenemos el resultado que queríamos probar.* ■

Capítulo 5

Concepto probabilístico de anomalía

Esta introducción de probabilidad nos va a servir tanto para la definición de anomalía alternativa a la basada en distancias que vamos a ver como para la explicación y análisis de los modelos.

En primer lugar cabe decir que esta definición, al igual que el criterio ya explicado no engloba todas las anomalías y por tanto es algo difícil de medir. Esta definición hace referencia, según mi criterio, a un enfoque que se debe poner junto a la definición basada en distancias y no en contraposición. El objetivo de esta definición es obtener anomalías que no son triviales y se esconden entre los datos.

La base del razonamiento de este tipo de anomalías surge del hecho de que un objeto puede ser anómalo en un subespacio concreto de los datos, pero no en el espacio total. Vamos a introducir un ejemplo para visualizar un tipo de anomalía que encaje con esta definición.

Veamos la siguiente figura:

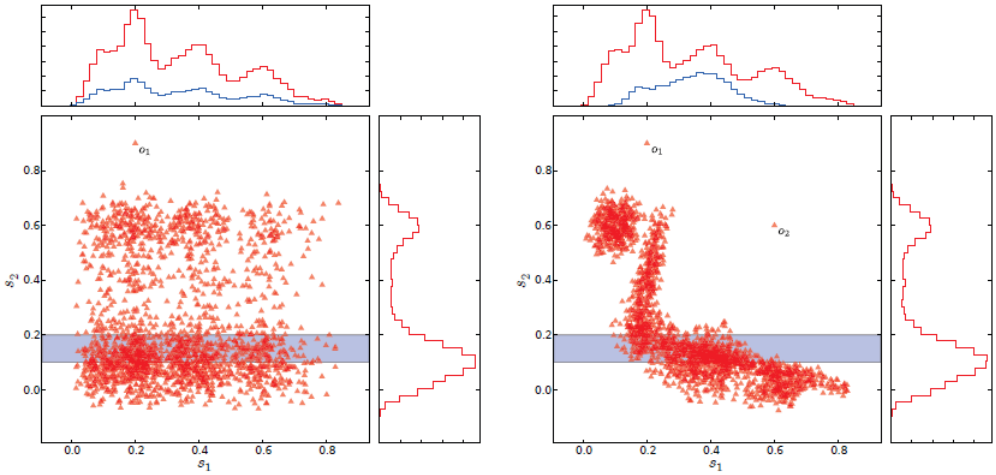

Figura 5.1: Ejemplo de anomalía [4]

Como se puede observar tenemos dos espacios: el izquierdo no presenta datos correlados y el derecho sí presenta correlación. Podemos ver que en ambos casos se comparte una anomalía etiquetada como O_1. Esta anomalía en el caso del espacio no correlado es perfectamente detectable de forma trivial observando las proyecciones de los datos en una dimensión. En cambio en el segundo caso ninguna de las dos anomalías etiquetadas O_1, O_2 son detectables de esta forma trivial, pues si hacemos las proyecciones uno dimensionales ninguno de los dos datos es discordante en dichas proyecciones. Estas anomalías son las que decimos que son no triviales. En cambio si observamos los datos en una proyección de orden superior como la que estamos viendo de dimensión 2 podemos observar claramente que se salen de la correlación de datos que muestra el resto. Es aquí donde podemos ver que en el conjunto de la derecha ninguno de los puntos es una anomalía en las proyecciones de dimensión uno pero sí lo son en la proyección de dimensión 2.

Vamos por tanto a definir más formalmente este concepto especial de anomalía. Necesitamos introducir en primer lugar un poco de notación.

Partimos de un conjunto de datos $X = \{x_1, ..., x_n\}$ de n objetos cada uno tomando d valores, es decir, $x_i = (x_{s_1}, ..., x_{s_d}) \in \mathbb{R}^d$. Notamos un subespacio del conjunto de valores como:

$$S = \{s_i | s_i \in \{s_1, ..., s_d\} \; con \; i \in \Delta\}$$

Dado un subespacio $S = \{s_1, ..., s_p\}$ notamos la proyección de los objetos del conjunto de datos como $X_S = \{x_{s_1}, ..., x_{s_p}\}$.

Esta proyección está distribuida según una distribución conjunta desconocida de S:

$$p_{s_1,...,s_p}(x_{s_1}, ..., x_{s_p})$$

Notamos la distribución marginal asociada al atributo s_i como:

$$p_{s_i}(x_{s_i})$$

Definición 5.1 *Decimos que un subespacio S es un espacio incorrelado si y sólo si:*

$$p_{s_1,...,s_p}(x_{s_1}, ..., x_{s_p}) = \prod_{i=1}^{p} p_{s_i}(x_{s_i})$$

Por tanto si estamos bajo la suposición de un espacio incorrelado podemos decir que la densidad esperada es:

$$p_{esp}(x_{s_1}, ..., x_{s_p}) \equiv \prod_{i=1}^{p} p_{s_i}(x_{s_i})$$

Recordemos que nuestras anomalías no triviales no están en este tipo de subespacios, si no en los correlados. Por tanto vamos a definirlo de la siguiente forma:

Definición 5.2 *Decimos que un objeto x_S es una anomalía no trivial respecto al subespacio S si:*

$$p_{s_1,...,s_p}(x_{s_1}, ..., x_{s_p}) \ll p_{esp}(x_{s_1}, ..., x_{s_p})$$

Es decir, si la probabilidad esperada es significativamente mayor que la probabilidad conjunta.

Por cómo hemos definido los espacios correlados e incorrelados es claro que no podemos tener anomalías en espacios no correlados como es evidente pues la densidad conjunta y esperada serían iguales.

Este concepto como podemos observar no comparte ninguna relación con nuestra definición de anomalías basadas en distancias por lo que es de esperar que si comparamos ambos tipos de anomalías en un conjunto de datos no obtengamos los mismos objetos.

Capítulo 6

Modelos implementados

En este capítulo vamos a repasar qué modelos he implementado y cómo funcionan cada uno de ellos. Primero se hará una revisión teórica de los modelos y posteriormente un análisis breve del código explicando las particularidades de las implementaciones.

6.1. Algoritmos de ensamblaje

Los algoritmos que he implementado pertenecen a una familia concreta de algoritmos de detección de anomalías denominados como algoritmos de ensamblaje o "Ensemble Algorithms" en inglés. Estos algoritmos son lo equivalente a los meta-algoritmos pero destinados a la detección de anomalías. Para dar una mejor definición de qué son los algoritmos de ensamblaje vamos a introducir una clasificación de los mismos para dar las categorías que entran dentro de esta definición.

- Algoritmos de ensamblaje secuenciales: En este tipo de algoritmos tenemos un algoritmos base o un conjunto de algoritmos base que se aplican de forma secuencial, de forma que las primeras ejecuciones se ven usadas o modificadas por ejecuciones futuras de algoritmos. Finalmente el resultado puede ser una combinación ponderada de las

valoraciones de los algoritmos o el resultado del último de ellos.

Ensamblaje secuencial:
Entrada: Conjunto de datos $\mathcal{D}$, Algoritmos base $\mathcal{A}_1, ..., \mathcal{A}_r$
j=1
repetir
  Tomamos el algoritmo $\mathcal{A}_j$ según los resultados anteriores
  Tomamos el conjunto de datos modificado $f_j(\mathcal{D})$ de anteriores ejecuciones
  Ejecutamos el algoritmo $\mathcal{A}_j$ sobre $f_j(\mathcal{D})$
  j=j+1
hasta que *fin*;
Resultado: Combinación de los resultados

- Algoritmos de ensamblaje independientes: En este caso se emplean o bien diferentes instancias del mismo algoritmo o bien diferentes porciones de los datos que se emplearán de forma distinta. Se puede variar la instanciación por ejemplo dependiendo del subespacio sobre el que queramos ejecutarlo o dependiendo de las características de una porción concreta de los datos.

  Ensamblaje independiente:
  Entrada: Conjunto de datos $\mathcal{D}$, Algoritmos base $\mathcal{A}_1, ..., \mathcal{A}_r$
  j=1
  repetir
    Tomamos el algoritmo $\mathcal{A}_j$
    Creamos el conjunto de datos modificado $f_j(\mathcal{D})$
    Ejecutamos el algoritmo $\mathcal{A}_j$ sobre $f_j(\mathcal{D})$
    j=j+1
  hasta que *fin*;
  Resultado: Combinación de los resultados

6.2. Mahalanobis Kernel

Este algoritmo está englobado dentro de la categoría de algoritmos basados en dependencia. Esta clase de algoritmos intenta estudiar las dependencias que existen entre atributos para así poder detectar las instancias u objetos que no tienen estas dependencias y marcarlos como anomalías.

Si intentamos visualizar esta dependencia entre atributos de forma gráfica lo que observaríamos es que los datos están alineados o posicionados en hiperplanos lineales o no lineales de la siguiente forma:

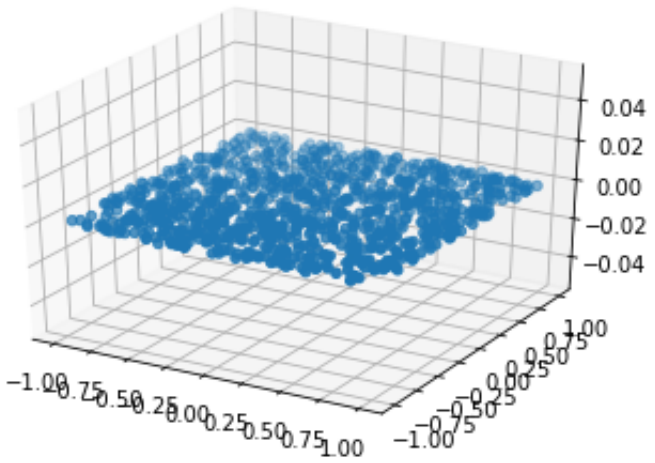

Figura 6.1: Hiperplano

Esta figura es un ejemplo clásico de estudio de algoritmos como por ejemplo PCA (algoritmo que quedaría dentro de esta categoría).

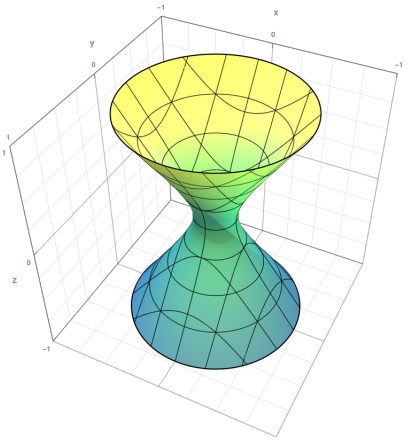

Figura 6.2: Hiperboloide Wikimedia

En este caso tenemos el ejemplo de un hiperboloide que no tiene una dependencia lineal, si no que presenta una dependencia de tipo cuadrático.

El método de Mahalanobis Kernel puede ser visto como una modificación de PCA. PCA básicamente dispone de dos pasos:

1. Determinar un sistema ortogonal de direcciones principales y proyectar los datos sobre este sistema.

2. Calcular la distancia entre el punto original y la proyección como su puntuación de anomalía.

El método Mahalanobis Kernel intenta tener este mismo comportamiento en dos pasos y que ahora veremos. El algoritmo PCA es muy útil cuando los datos tienen atributos relacionados en un hiperplano, mientras que Mahalanobis Kernel funciona mejor cuando los datos están relacionados en formas más complejas como el hiperboloide que hemos enseñado. La elección de este algoritmo en vez de PCA recae en el hecho de que PCA es un algoritmo clásico y el escenario en el que mejor funciona (hiperplano) es más restrictivo que el que nos ofrece Mahalanobis Kernel con un abanico de figuras más amplio.

Vamos a describir el funcionamiento del algoritmo, pero primero vamos a introducir notación. Vamos a llamar D a la matriz de datos que está centrada en la media y que tiene dimensiones $n \times d$, es decir, tenemos n instancias u objetos de dimensionalidad d.

Mahalanobis Kernel
Entrada: D
$S = DD^T$.
$S = Q\Delta^2 Q^T$.
Almacenamos los vectores propios columna no negativos de $Q\Delta$ en una matriz D'
Normalizamos D' para que tenga media 0 y varianza 1.
$vector\_media = media(D')$
$puntuaciones = []$
para cada *fila en D'* **hacer**
  $score = distancia(vector\_media, fila)$
  $puntuaciones = [puntuaciones, score]$
fin
Salida: puntuaciones

Algoritmo 1: Mahalanobis Kernel

El algoritmo comienza con la matriz de datos D. Se obtiene la matriz simétrica S y se hace la descomposición en valores singulares.

Con este modelo tenemos las dos fases que teníamos en PCA. Primero obtenemos una matriz D' de los datos proyectados y transformados para posteriormente reportar la puntuación de anomalía como una distancia.

Veamos ahora la implementación en Python.

```
1 def runMethod(self):
2     '''
```

```
3       @brief Function that executes the Kernel Mahalanobis
            method. The results are
4       stored on the variable self.scores
5       @param self
6       '''
7       ''' Compute the S matrix of the algorithm'''
8       S = np.dot(self.dataset, self.dataset.T)
9       ''' Now we diagonalize it'''
10      Q,delta_sq,Qt = np.linalg.svd(S)
11      del S
12      del Qt
13      ''' Obtain delta as matrix'''
14      delta = np.matrix(np.diag(np.sqrt(delta_sq)))
15      del delta_sq
16      Q = np.matrix(Q)
17      ''' Compute de D' matrix and normalize it'''
18      Dprime = np.dot(Q,delta)
19      del Q
20      del delta
21      Dp_std = scale(Dprime, axis=1)
22      del Dprime
23      ''' We compute its mean on the rows to compute the
            deviation as the score'''
24      mean = Dp_std.mean(axis=0)
25      self.outlier_score=[]
26      ''' The score is the euclidean distance to the mean'''
27      for i in range(len(Dp_std)):
28          self.outlier_score.append(np.linalg.norm(mean-
                Dp_std[i])**2)
29      self.outlier_score = np.array(self.outlier_score)
30      self.calculations_done=True
```

La implementación del algoritmo se ha realizado en Python como el resto del proyecto y posteriormente se explicará en detalle cómo se ha organizado.

El algoritmo basa su implementación en la librería NumPy.

6.3. TRINITY

Este algoritmo es del segundo tipo que vimos al principio cuando hicimos una categorización de los algoritmos de ensamblaje, en concreto el algoritmo hace una combinación de tres componentes distintos. La intención de hacer esta composición de modelos es intentar obtener todos los tipos de anomalías que se puedan del conjunto y que reciban una puntuación acorde. La teoría nos dice que esta combinación de modelos nos va a proveer de un resultado más robusto que el uso de modelos aislados como discutiremos en la sección

de resultados.

En concreto este algoritmo consta de tres componentes distintos:

- Componente basado en distancias: este componente consta de un algoritmo que base u comportamiento en técnicas de agrupamiento o valoración por distancias como por ejemplo es el método clásico KNN. Este método lo que hace es tomar los k vecinos más cercanos y colocar como puntaje de anomalía para esa instancia como la suma de estas distancias. De esta forma los puntos que más alejados estén del resto sumarán una mayor distancia y por tanto serán más anómalos. En concreto este modelo se ha utilizado con el valor $k = 5$ y con una técnica de subsampling. La técnica de subsampling consiste en no utilizar todo el conjunto de datos en el algoritmo, si no particionarlo y utilizar una pequeña muestra repitiendo este proceso y haciendo la media de las ejecuciones. De esta forma conseguimos una reducción de la varianza. Esto conlleva algunas ventajas como discutimos en la sección de sesgo y varianza anteriormente. En concreto la técnica toma 1000 particiones, ejecuta el algoritmo sobre ellas y hace la media.

- Componente basado en dependencia: este componente toma un algoritmo como el que hemos implementado (Mahalanobis Kernel). En este componente vamos a intentar detectar las anomalías que corresponden a datos que no siguen las relaciones entre atributos que sí tienen el resto de los objetos. Para ello he utilizado en este componente el algoritmo Mahalanobis Kernel que ya hemos explicado anteriormente incorporando la técnica de subsampling.

- Componente basado en densidad en subespacios: en este componente vamos a incorporar un modelo que intente buscar anomalías que lo sean en base a la densidad que tienen en alguno de los subespacios de los datos. Este hecho no nos debe ser ajeno pues es la segunda de las definiciones que hemos visto de anomalía y que hacía referencia a la función de densidad y los subespacios incorrelados y correlados. En concreto para este componente he utilizado el algoritmo IForest o Isolation Forest. Este algoritmo lo que hace es tomar de forma aleatoria un atributo y se van particionando los valores del mismo en una estructura de árbol, es decir, dividimos los datos en aquellos con un valor superior al marcado para el atributo y con un valor menor. De esta forma podemos medir cuántos pasos o lo que es lo mismo qué profundidad ha alcanzado nuestro árbol hasta llegar a dividir un objeto del resto de los datos. Este algoritmo también incorpora la técnica de subsampling.

Por último con esto hemos obtenido tres vectores o listas con la puntuación que cada componente nos ha arrojado para cada instancia. Para hacerlos comparables lo que debemos hacer es estandarizar los datos a media cero y varianza unitaria. Finalmente se realiza la media de los tres vectores de puntaje siendo esta la puntuación final devuelta por TRINITY.

Veamos la implementación de este algoritmo:

```
1  def distanceBased(self):
2      '''
3      @brief Function that implements the distance based
           component
4      @param self
5      @return It returns the vector with the scores of the
           instances
6      '''
7      ''' Initialize the scores'''
8      scores = np.array([0]*len(self.dataset)).astype(float)
9      for i in range(self.num_iter):
10         knn = KNN(n_neighbors=5, contamination=self.
               contamination)
11         ''' Number in the interval [50, 1000]'''
12         subsample_size = np.random.randint(50, 1001)
13         sample = []
14         if subsample_size>=len(self.dataset):
15             sample = list(range(len(self.dataset)))
16         else:
17             ''' Take the sample and train the model'''
18             sample = np.random.choice(len(self.dataset),
                   size=subsample_size, replace=False)
19         knn.fit(self.dataset[sample])
20         ''' Update the score to compute the mean'''
21         scores[sample]+=knn.decision_scores_
22     ''' Return the mean'''
23     scores = scores/self.num_iter
24     scores = scale(scores)
25     return scores
26
27 def dependencyBased(self):
28     '''
29     @brief Function that implements the dependency based
           component
30     @param self
31     @return It returns the vector with the scores of the
           instances
32     '''
33     ''' Initialize the scores'''
34     scores = np.array([0]*len(self.dataset)).astype(float)
35     for i in range(self.num_iter):
```

```
36          kernel_mahalanobis = KernelMahalanobis(
                contamination=self.contamination)
37          ''' Number in the interval [50, 1000]'''
38          subsample_size = np.random.randint(50, 1001)
39          sample = []
40          if subsample_size>=len(self.dataset):
41              sample = list(range(len(self.dataset)))
42          else:
43              ''' Take the sample and train the model'''
44              sample = np.random.choice(len(self.dataset),
                    size=subsample_size, replace=False)
45          kernel_mahalanobis.fit(self.dataset[sample])
46          ''' Update the score to compute the mean'''
47          scores[sample]+=kernel_mahalanobis.outlier_score
48      ''' Return the mean'''
49      scores = scores/self.num_iter
50      scores = scale(scores)
51      return scores
52
53  def densityBased(self):
54      '''
55      @brief Function that implements the dependency based
            component
56      @param self
57      @return It returns the vector with the scores of the
            instances
58      '''
59      ''' Initialize the scores'''
60      scores = np.array([0]*len(self.dataset)).astype(float)
61      for i in range(self.num_iter):
62          iforest = IForest(contamination=self.contamination
                , behaviour="new")
63          ''' Number in the interval [50, 1000]'''
64          subsample_size = np.random.randint(50, 1001)
65          sample = []
66          if subsample_size>=len(self.dataset):
67              sample = list(range(len(self.dataset)))
68          else:
69              ''' Take the sample and train the model'''
70              sample = np.random.choice(len(self.dataset),
                    size=subsample_size, replace=False)
71          iforest.fit(self.dataset[sample])
72          ''' Update the score to compute the mean'''
73          scores[sample]+=iforest.decision_scores_
74      ''' Return the mean'''
75      scores = scores/self.num_iter
76      scores = scale(scores)
77      return scores
78
```

```
79  def runMethod(self):
80      '''
81      @brief This function is the actual implementation of
            TRINITY
82      @param self
83      '''
84      ''' Distance module'''
85      if self.verbose:
86          print("Obtaining scores with the distance module")
87      distance_based = self.distanceBased()
88      ''' dependency module'''
89      if self.verbose:
90          print("Obtaining scores with the dependency module
                ")
91      dependency_based = self.dependencyBased()
92      ''' Density module'''
93      if self.verbose:
94          print("Obtaining scores with the density module")
95      density_based = self.densityBased()
96
97      ''' Compute the mean of the three modules'''
98      self.outlier_score=(distance_based + dependency_based
            + density_based)/3
99      self.calculations_done=True
```

Todos los módulos tienen una función parecida. En primer lugar se inicializan las puntuaciones y se repite el mismo proceso de cálculo 100 veces. Se inicializa el modelo y se ajusta con una muestra de tamaño en el intervalo $[50, 1000]$. Por último se hace la media de todos los cálculos y se estandarizan con la librería Sklearn.

En la función principal "runMethod" se ejecutan los tres módulos y se hace la media de las puntuaciones.

6.4. OUTRES

Este método entra dentro del segundo de los tipos que hemos visto en la clasificación inicial pues el objetivo es analizar los datos por subespacios. Una de las cosas que podremos ver al final cuando hagamos el estudio de los resultados es lo costoso de estos métodos, siendo este el primero que nos va a servir de ejemplo para visualizar este problema.

En primer lugar cabe decir que en el resto de algoritmos las puntuaciones reflejan el factor de anomalía en orden creciente, es decir, a mayor puntaje más anómalo es el dato y a menor puntaje menos anómalo se dice que es.

En este caso los puntajes van a estar en el intervalo $[0, 1]$ siendo 0 un puntaje para un dato lo más anómalo posible y 1 un puntaje para un dato lo más normal posible. Daremos razones para que esto sea así cuando veamos el algoritmo.

OUTRES
Entrada: o: instancia, S: subespacio
para cada $i \in (D \setminus S)$ **hacer**
  $S' = S \cup \{i\}$
  si S' *es relevate* **entonces**
    $den(o, S') = \frac{1}{n} \sum_{p \in AN(o,S')} K_e(\frac{dist_{S'}(o,p)}{\epsilon(|S'|)})$
    $dev(o, S') = \frac{\mu - den(o,S')}{2\sigma}$
    si $dev(o, S') \geq 1$ **entonces**
      $r(o) = r(o) \cdot \frac{den(o,S')}{dev(o,S')}$
    fin
    $OUTRES(o, S')$
  fin
  en otro caso
    Para recursividad
  fin
fin
Salida: r: puntajes

Algoritmo 2: OUTRES

En primer lugar tenemos que definir qué son los espacios relevantes. Decimos que un subespacio S es relevante si la proyección sobre ese subespacio no está distribuida de uniformemente. No podemos hacer un test de que toda la proyección esté distribuida uniformemente por lo que nos vamos a valer del siguiente teorema:

Teorema 6.1 *Sea S un subespacio del conjunto de datos. Si S está distribuido uniformemente entonces $\forall s_i \in S$ tenemos que la proyección unodimensional del conjunto de datos sobre s_i está distribuida uniformemente.*

Este teorema nos da la siguiente herramienta: si comprobamos que ninguna proyección uno-dimensional está distribuida uniformemente entonces podemos afirmar que S está distribuido uniformemente. Para hacer estas comprobaciones uno-dimensionales lo hemos hecho mediante el test de Kolmogorov-Smirnov.

Además el algoritmo incorpora un nuevo concepto. Los datos tienen mayor relevancia cuando los consideramos en subespacios y cuando los metemos dentro de lo que los autores llaman vecindarios adaptativos. La definición

de un vecindario adaptativo es la siguiente:

Definición 6.1 *Definimos para la instancia u objeto o en el subespacio S su vecindario adaptativo como:*

$$AN(o, S) = \{p | dist_S(o, p) \leq \epsilon(|S|)\}$$

Donde $dist_S$ es la función distancia sobre los atributos S. Aquí tenemos una distancia máxima definida en función del cardinal del subespacio que viene dada de la siguiente forma:

$$\epsilon(|S|) = 0.5 \cdot \frac{h_{optimal}(|S|)}{h_{optimal}(2)}$$

Donde:

$$h_{optimal}(d) = (\frac{8\Gamma(\frac{d}{2}+1)}{\pi^{\frac{d}{2}}}(d+4)(2\sqrt{\pi})^d)n^{\frac{-1}{d+4}}$$

Donde Γ denota la función gamma y n es el tamaño del conjunto de datos, es decir, el número de objetos o instancias.

Con esto ya tenemos la definición del vecindario adaptativo y el objeto de tenerlo es poder comprobar que el la instancia o es relevante en el subespacio S dentro de su vecindario adaptativo.

En cuanto a la definición de densidad tenemos la función K_ϵ que es el llamado Kernel de Epachenikov. Esta función está definida como:

$$K_\epsilon(x) = (1 - x^2) \ \forall x < 1$$

En cuanto al μ y σ que aparecen en el cálculo de la desviación son la media y la desviación típica de las densidades en el vecindario adaptativo de o en el subespacio S. Por tanto con la desviación estamos midiendo cómo de alejada está la instancia o en densidad respecto del resto de las instancias de su vecindario adaptativo. Si esta densidad es mayor a dos desviaciones típicas entonces $dev(o, S')$ será mayor que 1 y por tanto estaremos ante un dato anómalo.

Si este es el caso, al estar los puntajes de anomalías inicializados a 1 podemos actualizarlo multiplicando por un valor menor estricto que 1. En

este caso este valor es $\frac{den(o,S')}{dev(o,S')}$. Como este valor es menor que 1 reducirá el puntaje y lo acercará más a una anomalía. De esta forma cuantas más veces se actualize su puntaje y cuanto menor puntaje obtenga con $\frac{den(o,S')}{dev(o,S')}$ más anómalo consideraremos el dato.

Finalmente si el subespacio era relevante significa que aún podemos aumentar más la dimensionalidad pues puede que nos queden subespacios de mayor orden que sigan sin estar distribuidos según una uniforme.

Como dato cabe decir que el algoritmo empieza en dimensión 2 y no en dimensión 1 pues en una única dimensión no tiene sentido estudiar la densidad ya que no vamos a sacar información de calidad.

Veamos la implementación

```
1  def isRelevantSubspace(self, subspace, neighborhood):
2      '''
3      @brief Function that tells if a subspace is relevant,
           this is that the projection
4      of the dataset over the subspace is not distributed
           uniformly in the neighborhood
5      @param self
6      @param subspace Subspace to check
7      @param neighborhood neighborhood in which to check the
            projection
8      @return It returns True if the subspace is relevant,
           False in other case.
9      '''
10     ''' We check first if we have already considered this
           subspace. If so, it is not relevant anymore.'''
11     for sub in self.checked_subspaces:
12         if len(np.intersect1d(sub, subspace))==len(
               subspace):
13             return False
14
15     ''' Make the projection'''
16     projection = self.dataset[:,subspace][neighborhood]
17     if len(projection)==0:
18         return False
19     ''' We check for each subspace if the 1-dimensional
           data is uniformly distributed'''
20     for i in range(len(subspace)):
21         min = np.amin(projection[:,i].reshape(-1))
22         max = np.amax(projection[:,i].reshape(-1))
23         ''' We do it using the Kolmogorov-Smirnov test'''
24         d,p = kstest(projection[:,i], "uniform", args=(min
               , max-min))
25         ''' If the null hypothesis is not rejected, this
```

```
            means the data follow a uniform distribution
            '''
26          if p<=self.alpha:
27              return False
28      return True
29
30  def computeHOptimal(self, d):
31      '''
32      @brief Function that calculates the Hoptimal
33      @param self
34      @param d Parameter, usually the dimensionality of the
            subspace
35      @return It returns a numerical value.
36      '''
37      f1 = (8*gamma(d/2 + 1))/(np.power(np.pi, d/2))
38      f2 = d+4
39      f3 = np.power(2*np.sqrt(np.pi),d)
40      n = len(self.dataset)
41      f4 = np.power(n, -1/(d+4))
42      return f1*f2*f3*f4
43
44  def computeEpsilon(self, subspace):
45      '''
46      @brief Function to compute the epsilon of the
            adapatative neighborhood
47      @param self
48      @params subspace Subspace considered to compute the
            epsilon
49      @return It returns a numerical value
50      '''
51      return 0.5*(self.computeHOptimal(len(subspace))/self.
            computeHOptimal(2))
52
53  def computeNeighborhood(self, subspace, instance):
54      '''
55      @brief This function computes the adaptative
            neighborhood
56      @param subspace Subspace in which to compute the
            neighborhood
57      @param instance Instance considered as the centroid of
             the neighborhood (index of the element)
58      @return It returns a numpy array containing the
            indexes of the neighborhood
59      '''
60      # First we compute the projection
61      projection = self.dataset[:,subspace]
62      # We compute a numpy array of the distances of all the
             elements to the instance
63      tile = np.tile(projection[instance], len(self.dataset)
```

```
        ).reshape((len(self.dataset),len(projection[
        instance])))
64      distances = np.linalg.norm(projection-tile, axis=1)
65      # We keep only the ones that are close enough (epsilon
            distance as max)
66      neighborhood = np.where(distances<self.epsilons[len(
        subspace)])[0]
67      # We exclude the instance itself
68      return neighborhood[neighborhood!=instance]
69
70  def computeKernel(self, x):
71      '''
72      @brief Function that computes the Epanechnikov kernel
            with scalar factor 1
73      @param self
74      @param x Number between 0 and 1.
75      @return It returns a numerical value
76      '''
77      return 1-np.power(x,2)
78
79  def computeDensity(self, subspace, neighborhood, instance)
        :
80      '''
81      @brief This is the function that computes the density
82      @param self
83      @param subspace Subspace in which to compute the
            density
84      @param neighborhood Adaptative neighborhood for the
            instance in the subspace
85      @param instance Index of the instance considered at
            the moment
86      @return It return a numerical value.
87      '''
88      # Compute the projection
89      projection = self.dataset[:,subspace]
90      # Compute the density
91      tile = np.tile(projection[instance], len(self.dataset)
        ).reshape((len(self.dataset),len(projection[
        instance])))
92      return np.sum(self.computeKernel(np.linalg.norm(
        projection-tile, axis=1))/self.computeEpsilon(
        subspace))/len(self.dataset)
93
94  def computeDeviation(self, subspace, neighborhood,
        instance, density):
95      '''
96      @brief Function that computes the deviation
97      @param self
98      @param subspace Subspace considered to compute the
```

```
        deviation
99      @param neighborhood Adaptative neighborhood for the
        instance in the subspace
100     @param instance Instance to compute the deviation
101     @param density Density value of the instance
102     @return It returns a numerical value
103     '''
104     ''' First we need to compute the density for all the
        neighbors'''
105     densities = np.array([])
106     for neig in neighborhood:
107         local_neigborhood = self.computeNeighborhood(
            subspace, neig)
108         densities = np.append(densities,self.
            computeDensity(subspace, local_neigborhood,
            neig))
109     ''' We compute the mean and the standard deviation'''
110     mean = np.mean(densities)
111     stdv = np.std(densities)
112     ''' Return the deviation'''
113     return (mean-density)/(2*stdv)
114
115 def outres(self, instance, subspace):
116     '''
117     @brief Main loop of the outres algorithm
118     @param self
119     @param instance Instance to compute the outres score
120     @param subspace Initial subspace of dimension 1
121     '''
122     ''' First we compute the indexes of the features that
        are not used in the actual subspace'''
123     available_indexes = list(set(list(range(len(self.
        dataset[0])))).difference(set(list(subspace))))
124     ''' For each available index we are going to check'''
125     for index in available_indexes:
126         ''' We make the new subspace adding the index'''
127         new_subspace = np.append(subspace, int(index)).
            astype(int)
128         ''' We compute the adaptative neighborhood'''
129         neighborhood = self.computeNeighborhood(
            new_subspace, instance)
130         ''' If the subspace is relevant'''
131         if self.isRelevantSubspace(new_subspace,
            neighborhood):
132             ''' Compute the density and deviation'''
133             density = self.computeDensity(new_subspace,
                neighborhood, instance)
134             deviation = self.computeDeviation(new_subspace
                , neighborhood, instance, density)
```

```
135             ''' If it is a high deviating instance in the
                    subspace then we update the score'''
136             if deviation>=1:
137                 if self.verbose:
138                     print("The instance " + str(instance
                            +1) + " is outlying in the
                            subspace " + str(new_subspace))
139                 ''' The scores are equal to 1 at first and
                         1 means no outlierness and 0 means
                        very outlying'''
140                 self.outlier_score[instance]*=density/
                        deviation
141             ''' We keep the process if the subspace was
                    relevant'''
142             self.outres(instance, new_subspace)
143         ''' We add the subspace to the considered ones'''
144         self.checked_subspaces.append(new_subspace)
145
146
147 def runMethod(self):
148     '''
149     @brief This function is the actual implementation of
            OUTRES
150     '''
151     ''' First we compute all epsilons so we dont need to
            make this calculation more than once'''
152     self.epsilons = [self.computeEpsilon(list(range(n)))
            for n in range(len(self.dataset[0])+1)]
153
154     ''' We initialize the scores to one'''
155     self.outlier_score = np.ones(len(self.dataset))
156     ''' For each instance we run outres'''
157     for i in range(len(self.dataset)):
158         ''' Erase checked_subspaces'''
159         self.checked_subspaces = []
160         if self.verbose and i%25==0:
161             print("Computing the instance " + str(i+1) +
                        "/" + str(len(self.dataset)))
162         ''' We run for each instance each index'''
163         for j in range(len(self.dataset[0])):
164             self.outres(i,np.array([j]))
165     ''' At the end, score 1 means no outlierness and 0
            100% outlier. We make 1-score
166     so we can keep the ascending order and now this will
            mean that 0 is no outlierness
167     and 1 is very outlying.'''
168     self.outlier_score = np.ones(len(self.dataset))-self.
            outlier_score
169     self.calculations_done=True
```

6.5. HICS

HICS es otra aproximación distinta al estudio por subespacios como ha sido el algoritmo OUTRES. OUTRES intenta encontrar un subespacio interesante en el vecindario de una instancia, es decir, para cada instancia se busca el subespacio interesante para dicha instancia y no tiene por qué ser interesante para ninguna más. Esta aproximación es la opuesta pues la intención es buscar subespacios que sean interesantes en general y hacer una valoración de los datos sobre dichos subespacios.

Este algoritmo además pretende encontrar anomalías basándose en el concepto de densidad como introdujimos en la segunda definición de anomalía. Veamos primero el algoritmo en pseudocódigo para poder ir desgranándolo y explicarlo.

HICS
Entrada: D: dataset
$scores = [\ ]$
$sub =$ subespacios de alto contraste
para cada $S \in sub$ **hacer**
  Ajustamos un modelo con el algoritmo LOF con la proyección sobre S
  $scores = scores + puntaje\ LOF$
fin
$scores = \frac{scores}{|sub|}$
Salida: scores: puntajes

Algoritmo 3: HICS

Como podemos ver el objetivo de HICS es en primer lugar obtener una serie de subespacios que sean relevantes y que hemos llamado subespacios de alto contraste. Para cada uno de estos subespacios vamos a estudiar la proyección de los datos sobre estos subespacios y vamos a obtener una puntuación de las instancias con el modelo LOF. Finalmente hacemos la media de todos estos puntajes para obtener la puntuación final.

El modelo original está pensado para funcionar con LOF pero la teoría nos dice que podría funcionar con cualquier modelo basado en proximidad. Por tanto en la implementación desarrollada he incluido los modelos LOF, COF, CBLOF, LOCI, HBOS y SOD como alternativas entre las que elegir para la elección del modelo simple con el que obtener el puntaje.

Veamos ahora el pseudocódigo con el que obtenemos el contraste de un

subespacio.

CalcularConstraste
Entrada: subespacio: subespacio, M: número de iteraciones del subsampling, α: valor para obtener el tamaño de la muestra, D: conjunto de datos
$size = n \cdot \sqrt[|subespacio|]{\alpha}$
$dev = 0$
para cada $i \in [1, M]$ **hacer**
  $comp\_atr = aleatorio\ de\ subespacio$
  $sel\_obj =$ muestra aleatoria de D de tamaño $size$
  $dev = dev + CalcularDev(comp\_atr, sel\_obj, subespacio, D)$
fin
$dev = \frac{dev}{M}$
Salida: dev: contraste

Algoritmo 4: CalcularConstraste

En resumen lo que vamos a ir haciendo es aplicar una técnica de subsampling en el algoritmo. Vamos a hacer M ejecuciones para obtener el contraste tomando diferentes muestras de un tamaño fijado de antemano. Con estas muestras vamos a calcular la desviación del atributo de comparación *comp_atr* entre las instancias como vamos a ver ahora. Finalmente acumulamos toda esta desviación y obtenemos la media para ese subespacio. Con esto tenemos una medida de cuánto se desvían entre sí las instancias dentro de dicho subespacio.

CalcularDev
Entrada: $comp\_atr$: atributo con el que comparar, sel_obj: muestra seleccionada aleatoriamente, $subespacio$: subespacio sobre el que calcular la desviación, D: conjunto de datos
$max = 0$
para cada $d \in D$ **hacer**
  $cum_1 = \sum_{o \in D} o[comp\_atr]$ si $o[comp\_atr] < d[comp\_atr]$
  $cum_2 = \sum_{o \in sel\_obj} o[comp\_atr]$ si $o[comp\_atr] < d[comp\_atr]$
  $f_a = \frac{cum_1}{|D|}$
  $f_b = \frac{cum_2}{|D|}$
  $subs = |f_a - f_b|$
  si $subs > max$ **entonces**
    $max = subs$
  fin
fin
Salida: max: máxima desviación

Algoritmo 5: CalcularDev

Con este test lo que estamos haciendo es ver cómo se diferencia el atributo escogido para la comparación con cada instancia del resto del conjunto de datos. La idea es similar a la que expusimos cuando comentamos brevemente los Isolated Forests de ir dividiendo los datos con un valor de corte.

Ahora ya tenemos las herramientas con las que podemos medir el contraste de un subespacio. Para ver cuáles son aquellos con mayor contraste los autores probaron dos formas quedándose finalmente con la que yo he añadido a la implementación. En primer lugar podemos pensar en algún tipo de cota que se vaya adaptando al número de subespacios o al contraste que estos vayan presentando. Esta idea finalmente fue descartada primero por no funcionar de forma satisfactoria y segundo por la dificultad de elección de la cota. La opción escogida finalmente es evaluar para cada dimensión todos los subespacios posibles, se obtiene el contraste de cada uno de ellos y se toma un número fijo de los primeros. De esta forma obtendremos por ejemplo en cada dimensión 500 candidatos. Una vez que hayamos evaluado todas las dimensiones volvemos a ordenar los subespacios por contraste (ya sin tener en cuenta la dimensionalidad) y nos quedamos con los 1000 primeros. Con esta metodología vamos a tener la certeza de que vamos a quedarnos con los 1000 subespacios con mayor contraste de entre todos los posibles.

Veamos la implementación del algoritmo.

```
1  def computeContrast(self, subspace):
2      '''
3      @brief Function that computes the contrast for a given
           subspace
4      @param subspace Numpy array with the indexes of the
           features that define the subspace
5      @return It returns a float representing the contrast
           of the subspace
6      '''
7      ''' We set the adaptative size of the test'''
8      size = int(len(self.dataset)*np.power(self.alpha, len(
           subspace)))
9      ''' Number of instances in the dataset'''
10     N = len(self.dataset)
11     deviation = 0
12     ''' We repeat the process M times'''
13     for i in range(1,self.M+1):
14         ''' This is the comparison attribute for the test,
               so it will stay untouched'''
15         comparison_attr=np.random.randint(low=0, high=len(
               subspace))
16         ''' List of booleans that masks the instances of
               the dataset selected'''
```

```
17          selected_objects = np.array([True]*N)
18          ''' Select random indexes'''
19          selected_objects[reduce(np.union1d,np.array([np.
                random.choice(N,size=size,replace=False) for _
                 in range(len(subspace)-1)]))]=False
20          ''' With the sample given by the mask
                selected_objects we compute the deviation'''
21          deviation+=self.computeDeviation(subspace[
                comparison_attr], selected_objects, subspace)
22      ''' Finally the contrast is the average of all
            deviations'''
23      return deviation/self.M
24
25  def computeDeviation(self, comparison_attr,
        selected_objects, subspace):
26      '''
27      @brief Function that computes the deviation of the
            marginal distribution
28      given a fixed attribute, a sample and the condition
            given as a subspace
29      @param comparison_attr This is the comparison
            attribute
30      @param selected_objects Mask that sets a sample of the
             dataset
31      @param subspace Subspace or condition to calculate the
             deviation
32      @return It returns a float giving the deviation
33      '''
34      max = 0
35      ''' For each instance of the dataset'''
36      for d in self.dataset:
37          ''' This is the cumulative value for all elements
                in the dataset'''
38          cumul1 = np.sum(self.dataset[:,comparison_attr][
                self.dataset[:,comparison_attr]<d[
                comparison_attr]])
39          ''' This is the cumulative value for the
                selected_objects aka the sample'''
40          sel = self.dataset[:,comparison_attr][
                selected_objects]
41          cumul2 = np.sum(sel[sel<d[comparison_attr]])
42          ''' Finally we compute the average in both cases
                '''
43          fa = cumul1/len(self.dataset)
44          fb = cumul2/len(self.dataset)
45          ''' The difference in absolute value is the
                deviation'''
46          subs = np.absolute(fa-fb)
47          ''' We return the biggest of the deviations
```

```
            obtained'''
48          if subs>max:
49              max = subs
50      return max
51
52  def hicsFramework(self):
53      '''
54      @brief This function computes the high contrast
            subspaces on which to score the outlier
55      @return It returns a numpy array containing the high
            contrast subspaces
56      '''
57      ''' Ordered subspaces by dimension type: list of numpy
             arrays of numpy arrays
58       this means that in each positions there would be the
             subspaces of the corresponding
59       dimension in the form of a list of subspaces which
             are numpy arrays'''
60      all_subspaces = []
61      ''' Record of the contrast for each subspace in each
            dimension, same shape as all_subspaces'''
62      all_contrasts = []
63      ''' For all dimensions starting from dimension 2 (
            correlation has no sense on dimension 1)'''
64      for dimension in range(2,len(self.dataset[0])):
65          if self.verbose:
66              print("Computing subspaces in dimension " +
                    str(dimension) + "/" + str(len(self.
                    dataset[0])))
67          candidates = []
68          contrasts = []
69          ''' This list will keep the indexes of the
                redundant subspaces, those are d-dimensional
                subspaces with d+1-dimensional
70           subspaces containing them with higher contrast'''
71          redundant = []
72          ''' For dimension 2 we just obtain all possible
                indexes and make all combinations'''
73          if dimension==2:
74              ''' Calculate the candidates as all possible
                    combinations'''
75              indexes = list(range(len(self.dataset[0])))
76              candidates = np.array([np.array(list(comb))
                    for comb in list(combinations(indexes,
                    dimension))])
77              ''' Compute the contrasts'''
78              cont = 0
79              p = Pool(self.numThreads)
80              while cont+self.numThreads<len(candidates):
```

```
81                  contrasts = contrasts + p.map(self.
                        computeContrast,candidates[cont:cont+
                        self.numThreads])
82                  cont+=self.numThreads
83                  print("Computed " + str(cont) + "/" + str(
                        len(candidates)))
84              p = Pool(len(candidates)-cont)
85              contrasts = contrasts + p.map(self.
                    computeContrast,candidates[cont:])
86              print("Computed " + str(len(candidates)) + "/"
                     + str(len(candidates)))
87          else:
88              ''' We need to calculate now the indexes
                    starting from a previous subspace
89               We record the parent of each subspace to
                     check for redundancy'''
90              parents = []
91              ''' For all subspaces with one dimension less
                    '''
92              for i in range(len(all_subspaces[-1])):
93                  ''' We only consider new indexes, those
                        are the ones not uses in the father
                        subspace'''
94                  indexes = list(set(list(range(len(self.
                        dataset[0])))).difference(set(
                        all_subspaces[-1][i])))
95                  ''' For each new index'''
96                  for ind in indexes:
97                  ''' We calculate the new candidate as the
                        same subspace appending the index'''
98                  new_can = np.append(all_subspaces[-1][i],
                        ind)
99                  ''' Now we check that the candidate wasn't
                         in the list before'''
100                 new = True
101                 for previous in candidates:
102                     if len(np.intersect1d(previous,new_can
                            ))==len(new_can):
103                         new = False
104                 if new:
105                     candidates.append(new_can)
106                     parents.append(i)
107             ''' Compute the contrasts'''
108             cont = 0
109             p = Pool(self.numThreads)
110             while cont+self.numThreads<len(candidates):
111                 contrasts = contrasts + p.map(self.
                        computeContrast,candidates[cont:cont+
                        self.numThreads])
```

```
112                 cont+=self.numThreads
113                 print("Computed " + str(cont) + "/" + str(
                        len(candidates)))
114             p = Pool(len(candidates)-cont)
115             contrasts = contrasts + p.map(self.
                    computeContrast,candidates[cont:])
116             print("Computed " + str(len(candidates)) + "/"
                     + str(len(candidates)))
117 
118             ''' Check for redundancy'''
119             for i in range(len(parents)):
120                 if contrasts[i]>all_contrasts[-1][parents[
                        i]]:
121                     redundant.append(parents[i])
122 
123         candidates = np.array(candidates)
124         contrasts = np.array(contrasts)
125         ''' If there are redundant subspaces'''
126         if redundant!=[]:
127             if self.verbose:
128                 print("Now deleting redundant subspaces in
                         dimension " + str(dimension) + ", " +
                         str(len(redundant)) + " subspaces
                        removed.")
129         ''' Delete those ones'''
130         non_redundant_sub = np.delete(all_subspaces[-1],
                redundant)
131         ''' Update the subspaces'''
132         all_subspaces[-1]=non_redundant_sub
133         ''' Sort from higher contrast to lower and only
                get numCandidates number of subspaces if
                available'''
134         if len(candidates)>self.numCandidates:
135             all_subspaces.append(candidates[contrasts.
                    argsort()[-self.numCandidates:][::-1]])
136             all_contrasts.append(contrasts[contrasts.
                    argsort()[-self.numCandidates:][::-1]])
137         else:
138             all_subspaces.append(candidates)
139             all_contrasts.append(contrasts)
140     ''' We flatten the numpy array to obtain only a list
            of subspaces and contrasts'''
141     subspaces = np.array(all_subspaces).flatten()
142     contrasts = np.array(all_contrasts).flatten()
143     ''' We only give the maxOutputSpaces with higher
            contrast if available'''
144     if len(subspaces)>self.maxOutputSpaces:
145         return subspaces[contrasts.argsort()[-self.
                maxOutputSpaces:][::-1]]
```

```
146     return subspaces
147
148 def runMethod(self):
149     '''
150     @brief This function is the actual implementation of
            HICS
151     '''
152     if self.verbose:
153         print("Calculating the subspaces\n")
154     ''' First we obtain the high contrast subspaces'''
155     subspaces = self.hicsFramework()
156
157     if self.verbose:
158         print("Now calculating the scoring\n")
159     ''' We initialize the scores for each instance as 0'''
160     scores = np.zeros(len(self.dataset))
161     ''' For each subspace'''
162     for sub in subspaces:
163         ''' We place the corresponding scorer according to
                parameter'''
164         scorer = None
165         if self.outlier_rank=="lof":
166             scorer = LOF()
167         elif self.outlier_rank=="cof":
168             scorer = COF()
169         elif self.outlier_rank=="cblof":
170             scorer = CBLOF()
171         elif self.outlier_rank=="loci":
172             scorer = LOCI()
173         elif self.outlier_rank=="hbos":
174             scorer = HBOS()
175         elif self.outlier_rank=="sod":
176             scorer = SOD()
177         ''' Fits the scorer with the dataset'''
178         scorer.fit(self.dataset[:,sub])
179         ''' Adds the scores obtained to the global ones'''
180         scores = scores+scorer.decision_scores_
181     ''' Compute the average'''
182     self.outlier_score = scores/len(subspaces)
183     ''' Marks the calculations as done'''
184     self.calculations_done=True
```

6.6. LODA

LODA es un algoritmo que entra en la categoría de los algoritmos que involucran histogramas. Aún no hemos visto ningún algoritmo de este ti-

po, por lo que no hemos discutido el funcionamiento de estos algoritmos. Cuando hacemos un histograma de los datos estudiamos la distribución de probabilidad de los datos y por tanto podemos ver si éstos están en las colas en algún atributo o proyección o si están en el centro de la distribución. Con esto podemos saber si el dato es o no anómalo. En concreto LODA emplea una serie de proyecciones uno-dimensionales sobre las que se estudia la distribución.

En primer lugar vamos a ver cómo se obtienen los vectores que nos dan las proyecciones uno-dimensionales.

ProyeccionesAleatorias
Entrada: d: dimension, D: dataset, k: número de histogramas y proyecciones
$no\_neg = [\sqrt{d}]$
$proyecciones = [\ ]$
para cada $i \in [1, k]$ **hacer**
  $ind = no\_neg$ indices aleatorios en $[0, d]$
  $proy =$ vector con ceros en todas las posiciones menos en ind, donde hay valores sacados de una normal $\mathcal{N}(0, 1)$
  $proyecciones = [proecciones, proy]$
fin
Salida: $proyecciones$: proyecciones

Algoritmo 6: ProyeccionesAleatorias

En LODA tenemos un parámetro k que nos indica el número de proyecciones e histogramas que vamos a desarrollar. Esto nos va a dar más o menos muestras como haríamos con una técnica de subsampling tradicional.

Una vez que tenemos estos vectores de proyección vamos a ver cómo generamos los histogramas a partir de ellos.

ObtenerHistogramas
Entrada: D: dataset, $\{w_i\}_{i=1}^k$: vectores de proyecciones, k: numero de histogramas y vectores de proyección
Inicializamos los histogramas $\{h_i\}_{i=1}^k$
para $j = 1 \to |D|$ **hacer**
  para $i = 1 \to k$ **hacer**
    $z_i = x_j^T w_i$
    Actualiza el histograma h_i con z_i
  fin
fin
Salida: $\{h_i\}_{i=1}^k$: histogramas

Algoritmo 7: ObtenerHistogramas

Con esto ya tenemos tanto las proyecciones como los histogramas, por lo que sólo queda ver cómo obtenemos las puntuaciones de anomalías de los datos.

LODA
Entrada: x: instancia, $\{h_i\}_{i=1}^{k}$: histogramas, $\{w_i\}_{i=1}^{k}$: vectores de proyección
para $i = 1 \to k$ **hacer**
  $z_i = x^T w_i$
  Obtenemos $p_i = p_i(z_i)$ del histograma h_i
fin
$f = \frac{-1}{k} \sum_{i=1}^{k} \log(p_i(z_i))$
Salida: f: puntaje de anomalía de x

Algoritmo 8: LODA

Finalmente ya tenemos el algoritmo LODA completo. Cuando estamos evaluando una instancia obtenemos la probabilidad de que su proyección uno-dimensional ocurra. Con esta probabilidad si hacemos el logaritmo cuanto más cerca esté de 0 más cerca estará el valor de $-\infty$ y por tanto mayor va a ser nuestro puntaje anómalo de dicha instancia.

Ahora que ya tenemos el algoritmo completo vamos a ver la implementación:

```
1  def getRandomProjections(self, dimension):
2      '''
3      @brief Function that computes and returns the random
           projections
4      @param self
5      @param dimension Dimensionality of the dataset (int)
6      @return It returns a list with numpy arrays as
           projections
7      '''
8      ''' Number of non-negative elements in the projection
           '''
9      non_neg = int(np.ceil(np.sqrt(dimension)))
10     projections = []
11     ''' We are going to compute k projections'''
12     for i in range(self.k):
13         ''' Select non_neg random indexes to make the
               projection'''
14         ind = np.random.choice(dimension, replace=False,
               size=non_neg)
15         ''' Initialize it to zeroes'''
16         proj = np.zeros(dimension)
17         ''' The non-negative elements are drawn from a
               normal distribution'''
```

```
18          proj[ind]=np.random.normal(size=non_neg)
19          projections.append(proj)
20      return projections
21
22  def getBin(self, hist_limits, value):
23      '''
24      @brief Function that given a value, it returns the bin
            it belongs to for the histogram
25      @param self
26      @param hist_limits Limits for each bin of the
            histogram
27      @param value Value to check for the bin
28      @return It returns the index corresponding to the bin
29      '''
30      bin=-1
31      for i in range(len(hist_limits)):
32          if value<hist_limits[i]:
33              bin=i
34              break
35      return bin-1
36
37  def runMethod(self):
38      '''
39      @brief This is the implementation of the LODA
            algorithm
40      '''
41      ''' We compute first all projections'''
42      random_projections = self.getRandomProjections(len(
            self.dataset[0]))
43      ''' Initialize the histograms and the projected data
            '''
44      histograms = [[]]*self.k
45      Z = [[]]*self.k
46      ''' For each instance of the dataset'''
47      for j in range(len(self.dataset)):
48          ''' For each projection'''
49          for i in range(self.k):
50              ''' Compute the 1D projection'''
51              Z[i].append(np.dot(self.dataset[j].T,
                    random_projections[i]))
52      ''' Compute the k histograms with the data'''
53      for i in range(self.k):
54          histograms[i]=np.histogram(Z[i], bins = self.
                n_bins)
55
56      ''' Initialize the scores to zero'''
57      self.outlier_score = np.array([0]*len(self.dataset)).
            astype(float)
58      ''' For each instance'''
```

```
59      for i in range(len(self.dataset)):
60          prob = []
61          ''' For each histogram'''
62          for j in range(self.k):
63              ''' Compute the projection'''
64              z = np.dot(self.dataset[i].T,
                    random_projections[j])
65              ''' Check the bin for the projection'''
66              bin = self.getBin(histograms[j][1], z)
67              ''' Obtain the probability linked to z in the
                    histogram'''
68              prob.append(histograms[j][0][bin]/np.sum(
                    histograms[j][0]))
69          prob = np.array(prob)
70          ''' Compute the score with the probabilities'''
71          if 0. in prob:
72              self.outlier_score[i] = float("inf")
73          else:
74              self.outlier_score[i] = -np.sum(np.log(prob))/
                    self.k
75      self.calculations_done=True
```

6.7. Implementación

Todos los algoritmos están implementados utilizando una clase base llamada EnsembleTemplate:

```
1  class EnsembleTemplate:
2  '''
3  Template class for the ensemble anomaly detectors.
4  '''
5
6  def __init__(self, contamination=0.1):
7      '''
8      Init template
9      '''
10     pass
11
12 def fit(self, dataset):
13     '''
14     Function to set the dataset and execute the algorithm
15     '''
16     self.dataset = dataset
17     self.outlier_score = [0]*len(self.dataset)
18     self.outliers = []
19     self.runMethod()
```

```
20      return self
21
22  def runMethod(self):
23      '''
24      Function to run the method implemented
25      '''
26      pass
27
28  def getRawScores(self):
29      '''
30      Function that gets the raw scores
31      '''
32      return self.outlier_score
33
34  def getOutliersBN(self, noutliers):
35      '''
36      Function that gets the noutliers instances of the most
            outlying data
37      '''
38      return self.outliers
39
40  def getOutliers(self):
41      pass
```

El esqueleto de todas las clases que implementan los algoritmos de los que hemos hablado es este. En primer lugar todas las clases tienen un constructor en el que se pasan los parámetros de los modelos en caso de haberlos. En segundo lugar tenemos una función fit. El cometido de esta función es inicializar los puntajes, pasar el dataset al objeto de la clase para que se guarde y por último ejecutar el método que implementa.

La función principal es la función runMethod que es la que implementa la ejecución del algoritmo en sí. Esta función no está pensada para ser llamada externamente si no desde fit.

Por último tenemos tres funciones más. La primera de ellas nos da el vector que contiene los puntajes de las anomalías. La segunda función nos da los "noutliers" elementos con mayor puntaje y la tercera nos devuelve las instancias anómalas basándose en un parámetro que llamamos contaminación.

El parámetro de contaminación no es más que una estimación del porcentaje de anomalías que pensamos que va a tener el conjunto de datos. Por tanto si el parámetro de contaminación fuera por ejemplo 0.1 entonces esta función devolvería el primer 10 % con mayor valor de puntaje de anomalía.

Bibliografía

[1] Vapnik V. *The Nature of Statistical Learning Theory.* New York: Springer.

[2] Vladimir Cherkassky and Filip M. Mulier. *Learning from Data: Concepts, Theory, and Methods.* John Wiley & Sons. 02476.

[3] Hastie T., R. Tibshirani, and J. Friedman. *The elements of Statistical Learning: Data Mining Inference and Prediction.* New York: Springer.

[4] Fabian Keller, Emmanuel Müller, and Klemens Böhm. HiCS: High constrast subspaces for density-based outlier ranking. page 12.

[5] Abu-Mostafa Yaser, Magdon-Ismail Malik, and Lin Hsuan-Tien. *Learning from Data: a short course.*

[6] Barbora Micenková, Brian McWilliams, and Ira Assent. Learning representations for outlier detection on a budget. 00000.

[7] L. I. Kuncheva. A theoretical study on six classifier fusion strategies. 24(2):281–286. 00000.

[8] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera. A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. 42(4):463–484. 00000.

[9] S. Sukhanov, C. Debes, and A. M. Zoubir. Dynamic selection of classifiers for fusing imbalanced heterogeneous data. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5361–5365. 00000.

[10] Alessio Carrega, Francesca Cipollini, and Luca Oneto. Simple continuous optimal regions of the space of data. 00000.

[11] Francesca Cipollini, Luca Oneto, Andrea Coraddu, Alan John Murphy, and Davide Anguita. Condition-based maintenance of naval propulsion systems: Data analysis with minimal feedback. 177:12–23. 00005.

[12] Arthur Zimek, Matthew Gaudet, Ricardo JGB Campello, and Jörg Sander. Subsampling for efficient and effective unsupervised outlier detection ensembles. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 428–436. ACM.

[13] Charu C. Aggarwal and Saket Sathe. Theoretical foundations and algorithms for outlier ensembles. 17(1):24–47.

[14] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. 5(5):363–387.

[15] Aleksandar Lazarevic and Vipin Kumar. Feature bagging for outlier detection. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 157–166. ACM.

[16] Yue Zhao, Maciej K. Hryniewicki, Zain Nasrullah, and Zheng Li. LSCP: Locally selective combination in parallel outlier ensembles.

[17] Yue Zhao and Maciej K. Hryniewicki. XGBOD: improving supervised outlier detection with unsupervised representation learning. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

[18] Yue Zhao and Maciej K. Hryniewicki. DCSO: dynamic combination of detector scores for outlier ensembles. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), Outlier Detection De-constructed Workshop, London, UK*.

[19] Zengyou He, Shengchun Deng, and Xiaofei Xu. A unified subspace outlier ensemble framework for outlier detection. In *International Conference on Web-Age Information Management*, pages 632–637. Springer.

[20] Yue Zhao, Zain Nasrullah, and Zheng Li. PyOD: A python toolbox for scalable outlier detection.

[21] Alexander Strehl and Joydeep Ghosh. Cluster ensembles — a knowledge reuse framework for combining multiple partitions. 3:583–617.

[22] Charu C. Aggarwal and Philip S. Yu. Outlier detection for high dimensional data. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 37–46. ACM. event-place: Santa Barbara, California, USA.

[23] Emmanuel Müller, Matthias Schiffer, and Thomas Seidl. Statistical selection of relevant subspace projections for outlier ranking. In *Data*

Engineering (ICDE), 2011 IEEE 27th International Conference on, pages 434–445. IEEE.

[24] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. On detecting clustered anomalies using SCiForest. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 274–290. Springer.

[25] Ye Zhu and Kai Ming Ting. Commentary: a decomposition of the outlier detection problem into a set of supervised learning problems. 105(2):301–304.

[26] Patricia Iglesias Sánchez, Emmanuel Müller, Fabian Laforet, Fabian Keller, and Klemens Böhm. Statistical selection of congruent subspaces for mining attributed graphs. In *2013 IEEE 13th International Conference on Data Mining*, pages 647–656. IEEE.

[27] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. 6(1):3.

[28] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE.

[29] Jing Gao and Pang-Ning Tan. Converting output scores from outlier detection algorithms into probability estimates. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 212–221. IEEE.

[30] Tomáš Pevnỳ. Loda: Lightweight on-line detector of anomalies. 102(2):275–304.

[31] Saket Sathe and Charu C. Aggarwal. Subspace outlier detection in linear time with randomized hashing. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 459–468. IEEE.

[32] Leo Breiman. Bagging predictors. 24(2):123–140.

[33] Peter Lukas Bühlmann. Bagging, subagging and bragging for improving some prediction algorithms. In *Research report/Seminar für Statistik, Eidgenössische Technische Hochschule (ETH)*, volume 113. Seminar für Statistik, Eidgenössische Technische Hochschule (ETH), Zürich.

[34] Andreas Buja and Werner Stuetzle. Observations on bagging. 16(2):323.

[35] Peter Bühlmann and Bin Yu. Analyzing bagging. 30(4):927–961.

[36] Charu C. Aggarwal. Outlier ensembles: Position paper. 14(2):49–58.

[37] Shebuti Rayana and Leman Akoglu. Less is more: Building selective anomaly ensembles. 10(4):42.

[38] Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. Fast anomaly detection for streaming data. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

[39] Mahsa Salehi, Christopher A. Leckie, Masud Moshtaghi, and Tharshan Vaithianathan. A relevance weighted ensemble model for anomaly detection in switching data streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 461–473. Springer.

[40] Hoang Vu Nguyen, Hock Hee Ang, and Vivekanand Gopalkrishnan. Mining outliers with ensemble of heterogeneous detectors on random subspaces. In *International Conference on Database Systems for Advanced Applications*, pages 368–383. Springer.

[41] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC.

[42] Charu C. Aggarwal and Saket Sathe. *Outlier Ensembles: An Introduction*. Springer. Google-Books-ID: UNmfDgAAQBAJ.

[43] Charu C. Aggarwal. *Outlier Analysis*. Springer-Verlag.

[44] Obermayer. Proof of the key theorem of statistical learning theory.

[45] Juanjo Nieto and Antonia Delgado. Apuntes modelos matemáticos 2.

[46] Jürgen Braun and Michael Griebel. On a constructive proof of kolmogorov's superposition theorem. page 19.

[47] Hoang-Vu Nguyen, Emmanuel Müller, and Klemens Böhm. A near-linear time subspace search scheme for unsupervised selection of correlated features. 1:37–51. 00002.

[48] M. Loève. *Probability Theory*. Springer-Verlag.