Notas de clase

APRENDIZAJE DE MÁQUINAS

Versión: 13 de febrero de 2019

Felipe Tobar
Centro de Modelamiento Matemático
Universidad de Chile
ftobar@dim.uchile.cl

Prefacio

Este apunte es una versión extendida de las notas de clase del curso *MA5203: Aprendizaje de Máquinas Probabilístico* dictado en el Departamento de Ingeniería Matemática, Universidad de Chile. En este sentido, este apunte contiene estrictamente más contenidos que los vistos en clases en cada una de las unidades que componen el curso.

El fin último de este apunte es ser un libro autocontendio y original de las temáticas vistas en el curso arriba mencionado. Sin embargo, un objetivo importante es también apoyar el desarrollo del curso para que los alumnos puedan avanzar en los contenidos de éste de forma independiente en base a una fuente unificada. Por esta razón, esta versión preliminar del apunte está disponible para los alumnos del curso, en donde hay aún inconsistencias de formato, figuras o completitud de contenidos. En este sentido, este apunte será continuamnete actualizado.

Este esfuerzo ha sido posible gracias a la contribución de varios integrantes del cuerpo académico del curso MA5203 durante los años 2016, 2017 y 2018. Me gustaría reconocer la indispensable contribución de ayudantes del curso durante estos años, tanto en el desarrollo del curso mismo como en la escritura de secciones del apunte: Alejandro Cuevas, Gonzalo Ríos, Alejandro Veragua, Nicolás Aramayo, Lerko Araya, Mauricio Campos, Cristóbal Silva y Cristóbal Valenzuela. Sin la participación de todos ellos el éxito del curso y la composición de este apunte no habrían sido posible.

Felipe Tobar
13 de febrero de 2019
Santiago, Chile

Índice

1. Introducción

El aprendizaje de máquinas (AM) es una disciplina que reúne elementos de ciencias de la computación, optimización, estadística, probabilidades y ciencias cognitivas para construir el motor de aprendizaje dentro de la Inteligencia Artificial. Definido por Arthur Samuel en 1950, el AM es la disciplina que da a las máquinas la habilidad de aprender sin ser explícitamente programadas. Si bien existen enfoques al AM inspirados en sistemas biológicos, esta no es la única forma de construir métodos de aprendizaje: una analogía se puede identificar en los primeros intentos por construir máquinas voladoras, en donde se pretendía replicar el movimiento de las alas de un pájaro, sin embargo, estos intentos no fueron exitosos y no fue sino hasta la invención del primer avión, el cual no mueve sus alas, que el hombre logró construir la primera máquina voladora. Esto sugiere que el paradigma biológico no es exclusivo al momento de construir máquinas inteligentes que utilicen observaciones de su entorno para extraer información útil y generar predicciones.

AM es una disciplina joven que ha experimentado un vertiginoso crecimiento en las últimas décadas, esto ha sido posible en gran parte gracias a los recientes avances computacionales y la cantidad de datos que permite entrenar algoritmos complejos. El uso masivo de técnicas AM se ha visto reflejado en distintas áreas que incluyen visión computacional, clasificación de secuencias de ADN, marketing, detección de fraude, diagnósticos médicos, análisis financiero y traducción de texto por nombrar algunas. Adicionalmente, si bien el objetivo de AM es desarrollar algoritmos de aprendizaje prescindiendo en gran medida de la intervención humana, otra razón del éxito del AM es su facilidad para acoplarse con otras disciplinas aplicadas, en particular al área que hoy conocemos como *Data Science*.

1.1. Orígenes: Inteligencia Artificial

Nuestras habilidades cognitivas es lo que nos diferencia del resto de las especies del reino animal y el dominio del *Homo sapiens* sobre el planeta radica en estas habilidades: la inteligencia humana superior a la del resto de los animales nos permite adaptarnos a diferentes situaciones ambientales y sociológicas de forma rápida y sin la necesidad de un cambio evolutivo. Por ejemplo, para migrar desde África al norte de Europa y a Oceanía el Homo sapiens no se adaptó biológicamente a climas distintos, sino que manipuló herramientas y materiales para producir vestimenta adecuada y embarcaciones. Otra característica única del Homo sapiens es su habilidad de creer en un imaginario colectivo que permite construir organizaciones con un gran número de individuos, lo cual nuevamente exclusiva a nuestra especie y consecuencia de nuestra inteligencia (Harari, 2015). Como sociedad siempre hemos estado interesados en entender la inteligencia, en efecto, desde sus inicios la Filosofía y Sicología se han dedicado al estudio de las forma en que entendemos, recordamos, razonamos y aprendemos. La inteligencia artificial (IA) es una disciplina mucho más reciente que las anteriores y va un paso más allá de la mera comprensión de la inteligencia, pues apunta a replicarla y construir entes inteligentes (Russell & Norvig, 2003). Hay varias definiciones de IA dependiendo de si (i) adoptamos un punto de vista de razonamiento versus comportamiento, o bien si (ii) identificamos las acciones inteligentes como humanas u objetivamente racionales, una de estas posiciones es la de Alan Turing, el que mediante el juego de la imitación (Turing, 1950), sentencia que una máquina es inteligente si es capaz de desarrollar tareas cognitivas a un nivel "humano" suficiente para engañar a un interrogador (también humano). En este contexto para que una máquina sea inteligente, o equivalentemente, apruebe el test de Turing, es necesario que posea (Russell & Norvig, 2003):

- **procesamiento de lenguaje natural** para comunicarse con seres humanos, en particular,

el interrogador,

- **representación del conocimiento** para guardar información recibida antes y durante la interrogación,
- **razonamiento automático** para usar la información guardada y formular respuestas y conclusiones, y
- **aprendizaje de máquinas** para adaptarse a nuevas circunstancias y descubrir patrones.

El test de Turing sigue siendo un tópico de investigación en Filosofía hasta el día de hoy, sin embargo, los avances actuales de la inteligencia artificial no están necesariamente enfocados en diseñar máquinas para aprobar dicho test. Si bien los inicios de la IA están en la Filosofía, actualmente los avances en IA apuntan a desarrollar metodologías para aprender de grandes volúmenes de información sin necesariamente actuar de forma humana, en particular, el componente de aprendizaje de máquinas en la lista anterior ha jugado un papel fundamental en esta nueva etapa, en donde su aporte en distintas áreas de aplicación es cada vez más evidente.

1.2. Breve historia del aprendizaje de máquinas

En base a las definiciones de la inteligencia de máquinas asentadas por Turing en 1950, las primeras redes neuronales artificiales comenzaron a emerger en los trabajos seminales de (Minsky, 1952) que programó el primer simulador de una red neuronal, (Farley & Clark, 1954) que implementaron un algoritmo de prueba y error para el aprendizaje, y (Rosenblatt, 1958) que propuso el Perceptrón. En los años siguientes la investigación en redes neuronales se vio afectada por las limitaciones de dichas estructuras expuestas en (Minsky & Papert, 1969) dando origen a lo que es conocido como el primer invierno de la inteligencia artificial, en donde el Profesor Sir James Lighthill expuso frente al parlamento inglés que la inteligencia artificial se fijaba objetivos no realistas y solo servía para escenarios básicos (Lighthill, 1973). Esta desconfianza en los alcances de la IA ralentizó su desarrollo, sin embargo, los conexionistas seguirían investigando sobre formas de diseñar y entrenar redes neuronales, específicamente, los resultados de (Werbos, 1974) que culminarían en el algoritmo de backpropagation propuesto por (Rumelhart, Hinton, & Williams, 1986) y los avances de (Hopfield, 1982) en redes neuronales recurrentes permitirían terminar con el primer invierno de la inteligencia artificial.

Durante el receso del conexionismo proliferaron los sistemas basados en reglas, en particular, los sistemas experto compuestos por una serie de reglas condicionales "si-entonces" (if-then) que replican el comportamiento de un humano experto, estos métodos se convirtieron en la primera herramienta exitosa de la IA en aplicaciones reales. Sin embargo, los sistemas experto no aprenden por sí solos, en el sentido de que las reglas si-entonces deben ser explícitamente programadas por un humano, esto tiene un costo considerable dependiendo de la complejidad del problema, por lo que hacia el comienzo de la década de los 90s los sistemas experto colapsaron debido a que la cantidad de información disponible aumentaba y dicho enfoque no es "escalable". Un sistema basado en regla que aún se utiliza son los llamado árboles de decisión (Breiman, Friedman, Olshen, & Stone, 1984), los cuales difieren de los sistemas experto en que las reglas no son definidas por un humano sino que descubiertas en base a la elección de variables que mejor segmentan los datos de forma supervisada.

Las redes neuronales vieron un resurgimiento en los 80s con el método de backpropagation que permitía entrenar redes neuronales de más de dos capas usando la regla de la cadena, esto

permitió finalmente validar la premisa conexionista en tareas complejas, específicamente en reconocimiento de caracteres usando redes convolucionales como el Neocognitron (Fukushima, 1980) y LeNet-5 (LeCun, Boser, Denker, Henderson, R. E. Howard, & Jackel, 1989), y reconocimiento de voz usando redes neuronales recurrentes (Hopfield, 1982). Luego de esto vino una segunda caída del conexionismo hacia fines de los 80s, debido a inexistencia de una clara teoría que explicara el desempeño de las redes neuronales, su tendencia a sobreajustar y su elevado costo computacional. A principios de los 90s, y basado en la teoría del aprendizaje estadístico (Vapnik & Chervonenkis, 1971), surgieron los métodos basados en kernels, específicamente las máquinas de soporte vectorial (MSV) (Boser, Guyon, & Vapnik, 1992), esta nueva clase de algoritmos estaba fundamentada en una base teórica que combinaba elementos de estadística y análisis funcional para caracterizar los conceptos de sobreajuste, optimalidad de soluciones y funciones de costo en el contexto del aprendizaje de máquinas. Además de sus fundamentos teóricos, las MSV mostraron ser una alternativa competitiva a las redes neuronales en cuanto a su desempeño y costo computacional en distintas aplicaciones.

También en la década de los 90, surgieron nuevos enfoques de aprendizaje de máquinas en donde el manejo de incertidumbre era abordado usando teoría de probabilidades, este es probablemente es punto de mayor similitud entre AM y Estadística (Ghahramani, 2015). Este enfoque procede definiendo una clase de modelos generativos probabilísticos, es decir, se asume que los datos observados son generados por un modelo incierto y aleatorio, luego, el aprendizaje consiste en convertir la "probabilidad de los datos dado el modelo" en la "probabilidad del modelo dado los datos" mediante el teorema de Bayes. Este es un enfoque elegante y teóricamente sustentado que permitió reinterpretar enfoques anteriores, sin embargo, muchas veces la formulación probabilística resulta en que las estimaciones del modelo y las predicciones no tienen forma explícita, es por esto que para el éxito del AM bayesiano fue necesario recurrir a técnicas de inferencia aproximada basadas en Monte Carlo (Neal, 1993) o métodos variacionales (Jordan, Ghahramani, Jaakkola, & Saul, 1999). El enfoque bayesiano permite definir todos los elementos del problema de aprendizaje (modelos, parámetros y predicciones) mediante distribuciones de probabilidad con la finalidad de caracterizar la incertidumbre en el modelo y definir intervalos de confianza en las predicciones, esto incluso permite hacer inferencia sobre modelos con infinitos parámetros (Hjort, Holmes, Müller, & Walker, 2010). En estos casos, el espacio de parámetros pueden ser todas las posible soluciones de un problema de aprendizaje, por ejemplo, el conjunto de las funciones continuas en regresión (Rasmussen & Williams, 2006), o el conjunto de todas las distribuciones de probabilidad en el caso de estimación de densidades (Ferguson, 1973).

Un nuevo resurgimiento de las redes neuronales (RN) se vio en los primeros años de la década del 2000, donde el área se renombró deep learning (Bengio, 2009). Progresivamente, el foco de la comunidad migró desde temáticas probabilistas o basadas en kernels para volver a RN pero ahora con un mayor número de capas. El éxito del enfoque conexionista finalmente logró objetivos propuestos hace décadas principalmente por dos factores: (i) la gran cantidad de datos disponibles, e.g., la base de datos ImageNet, y (ii) la gran capacidad computacional y paralelización del entrenamiento mediante el uso de tarjetas gráficas, esto permitió finalmente implementar RNs con billones de parámetros, las cuales eran complementadas con técnicas para evitar el sobreajuste. El hecho que los parámetros pierden significado para el entendimiento de las relaciones entre los datos aleja al AM de la Estadística, donde el objetivo es netamente predictivo y no la inferencia estadística: hasta el momento no hay otro enfoque que supere a deep learning en variadas aplicaciones, esto ha sido principalmente confirmado por los avances de Google, DeepMind y Facebook. De acuerdo a Max Welling, si bien la irrupción de deep learning aleja al AM de la Estadística, aún hay temáticas que las se nutren de ambas áreas como programación probabilista y compu-

tación bayesiana aproximada (Welling, 2015), adicionalmente, Yoshua Bengio cree que aún hay muchos aspectos inciertos de deep learning en los cuales los estadísticos podrían ayudar, tal como los especialistas de las ciencias de la computación se han dedicado a los aspectos estadísticos del aprendizaje de máquinas en el pasado (Bengio, 2016)

1.3. Taxonomía del aprendizaje de máquinas

En la sección anterior se mencionaron distintos métodos de AM que son transversales a variados tipos de aprendizaje, en esta sección veremos los tres principales tipos de aprendizaje. Primero está el aprendizaje supervisado (AS), en donde los datos consisten en pares de la forma (dato, etiqueta) y el objetivo es estimar una función f(·) tal que etiqueta = f(dato) de acuerdo a cierta medida de rendimiento. El nombre supervisado viene del hecho que los datos disponibles están "etiquetados", y por ende es posible supervisar el entrenamiento del algoritmo. Ejemplos de AS son la identificación de spam en correos electrónicos (clasificación), como también la estimación del precio de una propiedad en función de su tamaño, ubicación y otras características (regresión) — donde ambas aplicaciones requieren de un conjunto de entrenamiento construido por un humano. La segunda categoría es el aprendizaje no supervisado (AnS), en donde los datos no están etiquetados y el objetivo es encontrar estructura entre ellos, es decir, agrupar subconjuntos de datos que tienen algún grado de relación o propiedades en común, por ejemplo, el clustering de un gran número de artículos en distintas categorías basado en su frecuencia aparición de palabras (Salakhutdinov, 2006). La tercera categoría del AM es el aprendizaje reforzado (AR), en donde un agente aprende a tomar decisiones mediante la maximización de un funcional de recompensa, este es probablemente el tipo de aprendizaje más cercano a la forma en que los animales aprendemos, mediante prueba y error, como por ejemplo cuando entrenamos un perro para que aprenda algún truco recompensándolo con comida cada vez realiza la tarea correctamente. El reciente resultado de DeepMind donde una máquina aprendió a jugar Go usando una búsqueda de árbol y una red neural profunda es uno de los ejemplos más exitosos de este aprendizaje reforzado (Silver, et al., 2016).

1.4. Relación con otras disciplinas

AM y Estadística. En el análisis de datos es posible identificar dos extremos: el aprendizaje inductivo en donde los datos son abundantes, los supuestos a priori sobre su naturaleza son vagos y el objetivo es realizar predicciones con algoritmos complejos que no necesariamente expliquen los datos. En el otro extremo está el aprendizaje deductivo, donde los datos no son masivos, se adoptan supuestos sobre su naturaleza y el objetivo es aprender relaciones significativas entre los datos usando modelos simples. Si bien en general al analizar datos nos movemos entre estos dos extremos, podemos decir que la estadística es más cercana al segundo extremo, mientras que el AM tiene componentes que son de enfoque deductivo y muy relacionado a estadística (inferencia bayesiana), y componentes de enfoque inductivo (redes neuronales). Una consecuencia de esto es la importancia que tienen los parámetros en distintos métodos de AM: En los métodos deductivos (estadísticos) los parámetros tienen un rol explicativo de la naturaleza del fenómeno en cuestión que es revelado por los datos, mientras que los métodos inductivos se caracterizan por tener una infinidad de parámetros sin una explicación clara, pues el objetivo muchas veces es directamente hacer predicciones. La relación entre estadística y algunos enfoques a AM es tan cercana que Robert Tibshirani se ha referido a AM como estadística pretenciosa (glorified statistics).

AM y Programación Clásica. La programación clásica construye algoritmos basados en reglas, es decir, una lista de instrucciones que son ejecutadas en una máquina, lo cual requiere que el programador conozca de antemano el algoritmo a programar, e.g., calcular la transformada de Fourier rápida (FFT) de una grabación de audio. Sin embargo, hay tareas en las cuales el algoritmo apropiado no es conocido, por lo tanto, el enfoque que adopta AM es programar directamente la máquina para que aprenda la lista de instrucciones. Tomemos el caso del ajedrez, de acuerdo a (Shannon, 1950) el número de combinaciones posibles para el juego de ajedrez es del orden de 1040, esto significa que usando programación clásica un programa ingenuo para jugar ajedrez tendría que tener al menos esa cantidad de instrucciones de la forma "para la combinación C, ejecutar la acción A". Si todos los humanos sobre la faz de la tierra se uniesen para programar esta rutina y cada uno pudiese escribir 10 de estas instrucciones por segundo, nos tomaría 4x1021 años, esto es casi un billón (1012) de veces la edad de la tierra (4.54x109), lo cual hace impracticable adoptar un enfoque clásico de programación. Una alternativa basada en AM es un programa simple en el cual la máquina explora distintos posibles escenarios del tablero e inicialmente toma decisiones aleatorias de qué acción ejecutar para luego registrar si dicha movida llevó a ganar o perder el juego, este enfoque de programación no pretende programar la máquina para "jugar ajedrez" sino para "aprender a jugar ajedrez". Una exitosa implementación de este concepto usando aprendizaje reforzado y redes neuronales profundas para el juego de Go puede verse en (Silver, et al., 2016).

AM, Knowledge Discovery in Databases (KDD) y minería de datos. KDD (Fayyad, Piatetsky-Shapiro, & Smyth, 1996) es "el proceso no trivial de identificar patrones potencialmente válidos, novedosos, útiles y explicativos en datos", y consta de 5 etapas: selección, preparación, transformación, minería e interpretación de datos. La etapa de minería de datos consiste en extraer información desde datos disponibles, por ejemplo, agruparlos en subconjuntos afines o identificar situaciones anómalas. Para esto se usan herramientas de AM, especialmente de aprendizaje no-supervisado debido a la cantidad de los datos, los cuales en general no están no etiquetados y existe poco conocimiento a priori de su naturaleza.

AM y Data Science. En línea con sus orígenes en la inteligencia artificial, el objetivo del AM es encontrar relaciones entre datos y hacer predicciones prescindiendo de la intervención humana, es decir, con poco o nada de conocimiento a priori. Sin embargo, las técnicas de AM pueden ser complementadas con el conocimiento de un problema específico, en donde especialistas en AM colaboran con especialistas de (i) el área en cuestión, (ii) minería de datos, y (iii) visualización. Esto es Data Science, una disciplina colaborativa donde especialistas de variadas áreas unen fuerzas para hacer un análisis detallado de los datos, con la finalidad de resolver un problema en particular, usualmente con fines comerciales. El perfil del Data Scientist es muy completo, pues debe tener conocimiento de AM, estadística, programación, minería de datos, interactuar con especialistas y entender el impacto comercial del análisis, de hecho, Data Scientist ha sido considerado el puesto de trabajo más sexy del siglo XXI según Harvard Business Review (Davenport & Patil, 2012).

1.5. Estado del aprendizaje de máquinas y desafíos

El aprendizaje de máquinas es una reciente disciplina que provee de herramientas a una gran cantidad de disciplinas, pero también es un área de investigación en sí misma con una activa comunidad y muchas preguntas abiertas. Desde el punto de vista algorítmico es posible identificar en primer lugar el desafío del entrenamiento en línea, es decir, ajustar el algoritmo cada vez que se dispone de un nuevo dato para operar continuamente (e.g. análisis de series de tiempo), este es un concepto fundamental en procesamiento adaptativo de señales y no ha sido tomado en cuenta satisfactoriamente aún en AM. Otro desafío que tiene relación con la implementación de algoritmos

es la capacidad de escalar el entrenamiento de AM para Big Data, pues en general los métodos de AM son costosos ya que se enfocan en descubrir aprendizaje y no en procesar datos de alta dimensión per se, sin embargo, esta habilidad es cada vez más necesaria en la era de la información donde, los conceptos que actualmente se usan en Big Data son básicos comparados con es estado del arte de AM.

También es posible identificar desafíos en un plano teórico, como por ejemplo la transferencia de aprendizaje, en donde la experiencia adquirida en la realización de una tarea (e.g., reconocimiento de automóviles) sirve como punto de partida para una tarea relacionada (e.g., reconocimiento de camiones), de la misma forma en que un físico o matemático es contratado para trabajar en un banco sin tener conocimiento a priori de finanzas. Por otro lado, un desafío ético son los riegos del uso de AM: el avance de AM parece a veces descontrolado y abre interrogantes con respecto de la legislación sobre el actuar de máquinas inteligentes. Por ejemplo, ¿quién es responsable en un accidente en el que está involucrado un automóvil autónomo? Estos últimos dos desafíos revelan que hay un gran área que no hemos explorado y que, a pesar de los avances teóricos y sobretodo aplicados del AM, estamos lejos de entender la inteligencia. Como ha sido expuesto en (Gal, 2015), nuestra relación con el entendimiento de la inteligencia mediante el uso del AM puede ser entendido como el Homo erectus hace 400.000 años frotando dos ramas para producir fuego, ellos usaron el fuego para abrigarse, cocinar y cazar, sin embargo, esto no quiere decir que entendían por qué al frotar dos ramas generaban fuego o, peor aún, qué es el fuego. Estamos en una etapa temprana del entendimiento del aprendizaje, en la que usamos estas herramientas "inteligentes" para nuestro bienestar, sin embargo, estamos lejos de entender la ciencia que hay detrás.

2. Regresión Lineal

El problema de regresión buscar determinar la relación entre una variable denominada *dependiente* (salida, respuesta o etiqueta) y una variable denominada *independiente* (entrada, estímulo o instancia). Intuitivamente, un modelo de regresión permite entender cómo cambia la variable dependiente cuando la variable dependiente es modificada. Esta relación entre ambas variables es representada por una función, consecuentemente, el problema de regresión es equivalente a encontrar una función. De esta forma, en base al espacio de posible funciones donde se busque dichas relación (por ejemplo el espacio de todos los polinomios de grado menor o igual a 5) y el criterio de búsuqeda que se aplique, se obtendrán distintos métodos de regresión.

El modelo básico de regresión, y que sirve de base para modelos más expresivos, es el de regresión lineal. En cuyo caso el espacio de funciones donde se busca la relación entre las variables dependientes e independienes es el de las funciones lineales afines.

Específicamente, para un conjunto de observaciones de entradas (x) y salidas (y) de la forma

$$\{(x_i, y_i)\}_{i=1}^{N} \subset \mathbb{R}^M \times \mathbb{R} \tag{1}$$

la regresión lineal busca encontrar un modelo lineal, es decir,

$$\begin{aligned} f\colon \mathbb{R}^M &\to \mathbb{R} \\ x &\mapsto f(x) = a^\top x + b, \quad a \in \mathbb{R}^M, b \in \mathbb{R} \end{aligned} \tag{2}$$

que represente de mejor forma el la dependencia de la variable y con respecto a la variable x en función de las observaciones en la ecuación (1).

2.1. Mínimos cuadrados

En este contexto aflora naturalmente la siguiente pregunta: ¿cuál es la mejor representación de los datos? o equivalentemente ¿como cuantificar la bondad de un modelo de regresión lineal? Un práctica ampliamente utilizada es elegir la función $f(\cdot)$ en la eq. (2) en base al criterio de mínimos cuadrados. Es decir, elegir los parámetros de la función $f(\cdot)$ que reporte un mínimo error cuadrático entre las observaciones en ecuación (1) y las predicciones calculadas por la función para los mismos datos.

Específicamente, el modelo linear de mínimos cuadrados está dado por

$$f^\star = \arg\min \frac{1}{2} \sum_{i=1}^{N} (y_i - f(x_i))^2 \tag{3}$$

el cual en el caso lineal es equivalente a encontrar los parámetros a y b dados por

$$a^\star, b^\star = \arg\min \frac{1}{2} \sum_{i=1}^{N} (y_i - a^\top x_i + b)^2 \tag{4}$$

Observe que la ecuación anterior es cuadrática en a y b, por lo cual tiene un único mínimo que puede ser encontrado explícitamente. Para esto, como la función f en la ecuación (2) no es lineal sino que *afín*, hacemos el siguiente cambio de variable:

$$\begin{bmatrix} x \\ 1 \end{bmatrix} \mapsto \tilde{x} \in \mathbb{R}^{M+1}, \quad \begin{bmatrix} a \\ b \end{bmatrix} \mapsto \theta \in \mathbb{R}^{M+1} \tag{5}$$

con lo cual el funcional cuadrático a minimizar se convierte en

$$J = \frac{1}{2}\sum_{i=1}^{N}(y_i - \theta^\top \tilde{x}_i)^2 \tag{6}$$

y el parámetro θ de mínimos cuadrados puede ser encontrado aplicando las condiciones de primer orden de la siguiente forma:

$$\begin{aligned}
\nabla J = 0 \Leftrightarrow & \sum_{i=1}^{N}(y_i - \theta^\top \tilde{x}_i)\tilde{x}_i^\top = 0 && \text{def. } J \\
\Leftrightarrow & \sum_{i=1}^{N} y_i \tilde{x}_i^\top = \sum_{i=1}^{N} \theta^\top \tilde{x}_i \tilde{x}_i^\top && \text{ordernando} \\
\Leftrightarrow & \theta^\top = \sum_{i=1}^{N} y_i \tilde{x}_i^\top \left(\sum_{i=1}^{N} \tilde{x}_i \tilde{x}_i^\top\right)^{-1} && \text{despejar } \theta^\top \\
\Leftrightarrow & \theta = \left(\sum_{i=1}^{N} \tilde{x}_i \tilde{x}_i^\top\right)^{-1} \sum_{i=1}^{N} \tilde{x}_i y_i && \text{transponer} \\
\Leftrightarrow & \theta = \left(\tilde{X}^\top \tilde{X}\right)^{-1} \tilde{X}^\top Y && \text{def. } \tilde{X} \text{ y } Y
\end{aligned} \tag{7}$$

donde $\tilde{X}$ y Y son las matrices de datos definidas por

$$\tilde{X} = \begin{bmatrix} \tilde{x}_1^\top \\ \vdots \\ \tilde{x}_N^\top \end{bmatrix} \in \mathbb{R}^{N \times M+1}, \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^N \tag{8}$$

La Figura 1 muestra un ejemplo de regresión lineal para observaciones de chirridos por segundo en función de la temperatura.

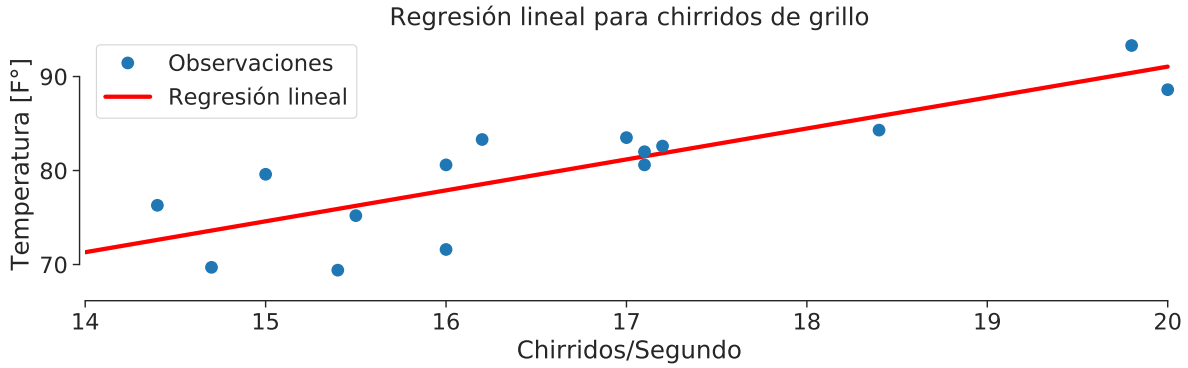


Figura 1: Ejemplo de regresión lineal.

La expresión $\left(\tilde{X}^\top \tilde{X}\right)^{-1} \tilde{X}^\top$ en la ecuación (7) es conocida como la pseudo-inversa de Moore-Penrose. Nótese que ésta solo existe si se tienen más observaciones (N) que dimensiones ($M+1$) y además si las observaciones no son colineales. En la práctica, es usual que tengamos más observaciones que parámetros, sobretodo en el caso lineal, sin embargo, es posible que las observaciones

sean redundantes, en cuyo caso la inversa de Moore-Penrose puede ser *cercana a no invertible*. Esto puede llevar a problemas de inestabilidad numérica al tratar de calcular su inversa. Una forma de solucionar esto penalizar soluciones que son *cercanas a ser no invertibles* o *irregulares*, de forma similar que se penalizan funciones que no representan bien los datos, en el sentido del error cuadrático. En este caso decimos que estamos *regularizando* la solución.

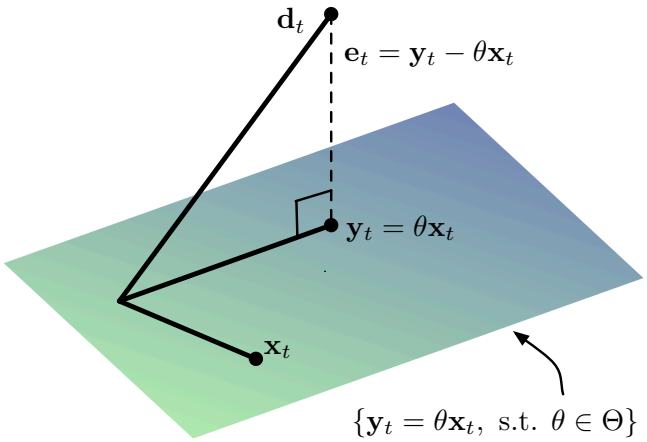


Figura 2: Interpretación geométrica de la regresión lineal

2.1.1. Mínimos cuadrados regularizados

El criterio para ajustar los parámetros de la regresión lineal puede, además de incluir el ajuste de los datos incluir ciertas penalizaciones sobre los modelos a elegir. Estas penalizaciones pueden ser codificadas directamente en la función de costo, donde ésta tiene ahora un término que penaliza mal ajuste y otro término que penaliza soluciones que se alejan de lo deseado. Un criterio estándar de penalización es la norma de los parámetros, es decir,

$$J_r = \frac{1}{2}\sum_{i=1}^{N}(y_i - \theta^\top \tilde{x}_i)^2 + \frac{\rho}{p}||\theta||_p^p,\ p \in \mathbb{R}_+ \tag{9}$$

donde $||\cdot||_p$ denota la norma ℓ_p, es decir, $||\theta||_p = (\sum_{j=1}^{N}\theta_j^p)^{\frac{1}{p}}$. Distintos valores de p inducen distintos propiedades sobre las soluciones, siendo las más usadas las correspondientes a $p = 1$ (lasso) o $p = 2$ (Tikhonov/Ridge). El parámetro ρ sirve para balancear la importancia entre ajuste y regularidad.

Para $p = 2$ tenemos que $||\theta||_2 = \theta^\top\theta$, con lo que el minimizador de la función de costo está dado por:

$$\nabla J_r = 0 \Leftrightarrow \sum_{i=1}^{N}(\theta^\top \tilde{x}_i - y_i)\tilde{x}_i^\top + \rho\theta^\top = 0 \qquad \text{def. } J \tag{10}$$

$$\Leftrightarrow \sum_{i=1}^{N} y_i\tilde{x}_i^\top = \sum_{i=1}^{N} \theta^\top \tilde{x}_i\tilde{x}_i^\top + \rho\theta^\top \qquad \text{ordernando} \tag{11}$$

$$\Leftrightarrow \theta^\top = \sum_{i=1}^{N} y_i\tilde{x}_i^\top \left(\sum_{i=1}^{N} \tilde{x}_i\tilde{x}_i^\top + \rho\mathbb{I}\right)^{-1} \qquad \text{despejar } \theta^\top \tag{12}$$

$$\Leftrightarrow \theta = \left(\sum_{i=1}^{N} \tilde{x}_i\tilde{x}_i^\top + \rho\mathbb{I}\right)^{-1} \sum_{i=1}^{N} \tilde{x}_i y_i \qquad \text{transponer} \tag{13}$$

$$\Leftrightarrow \theta = \left(\tilde{X}^\top\tilde{X} + \rho\mathbb{I}\right)^{-1} \tilde{X}^\top Y \qquad \text{def. } \tilde{X} \text{ y } Y \tag{14}$$

Observaciones:

1) Lasso vs Ridge ¿Qué busca cada uno? Habilidad de selección de variable con Lasso.

——— clase 22/3/18 ———

2.1.2. Máxima verosimilitud

Ahora tomaremos un enfoque distinto, en el cual no buscaremos el modelo que menor discrepancia tiene con los datos, sino que vamos a elegir el modelo que *más probablemente* haya generado los datos. La diferencia fundamental en este caso es que necesitamos diseñar un modelo que generó exactamente los datos, lo cual requiere modelar el ruido de las observaciones.

Consideremos el siguiente modelo

$$y = \theta^\top \tilde{x} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2) \text{ i.i.d.} \tag{15}$$

el cual representa la variable dependiente y mediante una parte determinística que depende linealmente de $\tilde{x}$ y una parte aleatoria. El modelo probabilístico en la ecuación (15) puede expresarse en forma distribucional como

$$y|\tilde{x} \sim p(y|\tilde{x}, \theta) = \mathcal{N}(y; \theta^\top\tilde{x}, \sigma_\epsilon^2). \tag{16}$$

Adicionalmente, como las observaciones son *condicionalmente independiente dado el modelo*, la probabilidad de todo el conjunto de entrenamiento $T = \{(x_i, y_i)\}_{i=1}^N$ es factorizable y dada por

$$p(Y|\tilde{X}, \theta) = p(y_1, \ldots, y_N|\tilde{x}_1, \ldots, \tilde{x}_N, \theta) = \prod_{i=1}^{N} p(y_i|\tilde{x}_i, \theta) = \prod_{i=1}^{N} \mathcal{N}(y_i; \theta^\top\tilde{x}_i, \sigma_\epsilon^2). \tag{17}$$

El ajuste del modelo entonces puede hacerse mediante la maximización (con respecto a θ) de la probabilidad que dicho modelo haya generado los datos de entrenamiento en la ecuación (1). Es decir:

$$\theta^\star = \arg\max p(y_1, \ldots, y_N|\tilde{x}_1, \ldots, \tilde{x}_N, \theta) \tag{18}$$

en el caso de arriba, las observaciones son condicionalmente independientes, y su distribución está dada por la ecuación (15). Consecuentemente:

$$\theta^\star = \arg\max \prod_{i=1}^{N} p(y_i|\tilde{x}_i, \theta) \tag{19}$$

$$= \arg\max \prod_{i=1}^{N} \mathcal{N}(y_i; \theta^\top \tilde{x}_i, \sigma_\epsilon^2) \tag{20}$$

$$= \arg\max \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_\epsilon} \exp\left(\frac{-1}{2\sigma_\epsilon^2}(y_i - \theta^\top \tilde{x}_i)^2\right) \tag{21}$$

$$= \arg\min \sum_{i=1}^{N} (y_i - \theta^\top \tilde{x}_i)^2 \tag{22}$$

La misma solución exacta que mínimos cuadrados, la diferencia es que también podemos maximizar con respecto a la varianza del ruido.

clase 27/3/18

2.1.3. Regresión Bayesiana

Repaso probabilidades - pending

Tres pasos de inferencia Bayesiana

- Definir un modelo conjunto para todas las cantidades involucradas, observaciones, valores futuros, parametros, etc
- Ajustar el modelo a la luz de observaciones
- Evaluar el modelo e interpretar resultados

En vez de maximizar la probabilidad de que los datos hayan sido generado por el modelo propuesto, podemos calcular la distribución condicional sobre modelos dado el conjunto de observaciones, es decir, $p(\theta|T)$. Esta es conocida como *distribución posterior del modelo dado los datos* y mediante el teorema de Bayes puede expresarse como

$$p(\theta|T) = \frac{p(Y|\theta, \tilde{X})p(\theta, \tilde{X})}{p(Y|\tilde{X})} \tag{23}$$

Observaciones:

1) La constante de normalización es en general muy complicada de calcular, pues es la probabilidad de que la salida sea Y para una entrada $\tilde{x}$ y *cualquier* modelo, de hecho,

$$p(Y|\tilde{X}) = \int p(Y, \theta|\tilde{X}) \mathrm{d}\theta = \int p(Y|\theta, \tilde{X}) p(\theta) \mathrm{d}\theta \tag{24}$$

Sin embargo, nótese que $p(Y|\tilde{X})$ es solo una constante de normalización y como $p(\theta|T)$ es una función de θ, el denominador en ecuación (23) solo amplifica dicha función. Es decir

$$p(\theta|T) \propto p(Y|\theta, \tilde{X})p(\theta, \tilde{X}) \tag{25}$$

La distribución posterior es entonces una *mezcla* entre la distribución posterior (que representa la creencia en la variable antes de ver datos) y la verosimilitud (que representa la probabilidad de los datos condicional al modelo). Cuando la distribución a priori y a posteriori son de la misma familia, diremos que son distribuciones conjugadas y además que el prior es un *prior* conjugado para la verosimilitud. Ejemplos de priors conjugados son las distribuciones gaussianas, por ejemplo, si consideramos $p(\theta, \tilde{X}) = \mathcal{N}(0, \sigma_\theta^2)$ y una verosimilitud gaussiana (i.e., modelo linear con ruido gaussiano), tenemos

$$\begin{aligned} p(\theta|T) &\propto p(Y|\theta, \tilde{X})p(\theta, \tilde{X}) && (26)\\ &= \mathcal{N}(Y; \theta^\top \tilde{X}, \mathbb{I}\sigma_\epsilon^2)\mathcal{N}(\theta; 0, \sigma_\theta^2) && (27)\\ &= \mathcal{N}(\theta; \mu, \Sigma) && (28) \end{aligned}$$

Observaciones:
1) ¿y la constante de normalización?

Ejemplo:
Otro caso de priors conjugados es la distribución binomial, que consiste en la probabilidad de obtener s aciertos en n intentos, cada uno independientes y equiprobables con probabilidad desconocida q.

$$p(n,s) = \binom{n}{s} q^s (1-q)^{n-s} \tag{29}$$

Donde su prior conjugado es la distribución Beta con parámetros (α, β)

$$p(q) = \frac{q^{\alpha-1}(1-q)^{\beta-1}}{\mathcal{B}(\alpha,\beta)} \tag{30}$$

Donde $\mathcal{B}(x,y)$ es la función Beta que actua como factor normalizador.
Luego, si queremos obtener la posterior de q con un prior Beta,

$$\begin{aligned}
p(q|n,s) &= \frac{p(n.s|q)p(q)}{p(n,s)} && (31)\\
&= \frac{\binom{n}{s} q^s (1-q)^{n-s} q^{\alpha-1}(1-q)^{\beta-1}}{p(n,s)} && (32)\\
&= \frac{\binom{n}{s}}{\underbrace{p(n,s)}} q^s (1-q)^{n-s} q^{\alpha-1}(1-q)^{\beta-1} && (33)\\
&= \mathcal{B}(s+\alpha, \beta+n-s) q^{s+\alpha-1}(1-q)^{n-s+\beta-1} \qquad \text{Es una distribución Beta} && (34)
\end{aligned}$$

Aplicación:
Se sabe que el 1 % de las mujeres tienen cancer de mamas, y se tienen un test para detectar si una mujer lo presenta o no. Si la paciente tiene cancer (C), el test dará postitivo (PT) con una probabilidad del 80 % y negativo (NT) con 20 %, en cambio cuando la paciente está sana (NC), hay un 9.6 % de probabilidad que el test salga erroneo y si detecte cancer.
Una paciente se realiza el test y este sale positivo, nos gustaría obtener la probabilidad de que en realidad tenga cancer dado este resultado.

$$\begin{aligned}
p(C|PT) &= \frac{p(PT|C)p(C)}{p(PT)} && (35)\\
&= \frac{p(PT|C)p(C)}{p(PT|C)p(C) + p(PT|NC)p(NC)} && (36)\\
&= \frac{0.8 \cdot 0.001}{0.8 \cdot 0.01 + 0.096 \cdot 0.99} && (37)\\
&= 0.0776 && (38)
\end{aligned}$$

De esta misma forma podemos completar todos los casos.

	C (1 %)	NC(99 %)
PT	0.8	0.096
NT	0.2	0.904

Figura 3: Resumen de las probabilidades para el caso del test de cancer.

2.2. Maximo a posteriori

¿Cuál es el equivalente a regularizar en es sentido probabilístico? La respuesta está en asignar más probabilidad *a priori* antes de ver datos en vez de penalizar y encontrar (una) moda de la distribución posterior, es decir

$$\theta^{\star}_{\text{MAP}} = \arg\max p(Y|\theta, \tilde{X})p(\theta, \tilde{X}) \tag{39}$$

$$= \arg\max \prod_{i=1}^{N} p(y_i|\tilde{x}_i, \theta)p(\theta, \tilde{X}) \tag{40}$$

$$= \arg\max \prod_{i=1}^{N} \mathcal{N}(y_i; \theta^{\top}\tilde{x}_i, \sigma^2_{\epsilon})\mathcal{N}(\theta; 0, \sigma^2_{\theta}) \tag{41}$$

$$= \arg\max \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_{\epsilon}} \exp\left(\frac{-1}{2\sigma^2_{\epsilon}}(y_i - \theta^{\top}\tilde{x}_i)^2\right) \frac{1}{(\sqrt{2\pi}\sigma_{\theta})^{M+1}} \exp\left(\frac{-||\theta||^2}{2\sigma^2_{\theta}}\right) \tag{42}$$

$$= \arg\max \frac{1}{\sqrt{2\pi}\sigma_{\epsilon}} \frac{1}{(\sqrt{2\pi}\sigma_{\theta})^{M+1}} \exp\left(\sum_{i=1}^{N} \frac{-1}{2\sigma^2_{\epsilon}}(y_i - \theta^{\top}\tilde{x}_i)^2 - \frac{||\theta||^2}{2\sigma^2_{\theta}}\right) \tag{43}$$

$$= \arg\min \sum_{i=1}^{N} (y_i - \theta^{\top}\tilde{x}_i)^2 + \frac{\sigma^2_{\epsilon}}{\sigma^2_{\theta}}||\theta||^2 \quad (:|) \tag{44}$$

2.3. Predicciones

Una vez que el modelo está ajustado, ya sea con estimaciones paramétricas puntiales o distribucionales, ¿cómo hacemos predicciones? En el caso puntual, la predicción es simplemente evaluar el modelo en el parámetro encontrado θ y la entrada x. En el caso de la estimación bayesiana del parámetro, la predicción toma en cuenta todos los posibles valores del parámetro. En efecto, la distribución de la variable dependiente y para una nueva entrada x está dada por la distribución condicional del y dado todos los datos de entrenamiento, es decir,

$$p(y|x, \{(x_i, y_i)\}_{i=1}^{N}) = \int p(y|x, \theta)p(\theta|\{(x_i, y_i)\}_{i=1}^{N})d\theta \tag{45}$$

consecuentemente, la esperanza está dada por

$$\mathbb{E}\, y|x, \{(x_i, y_i)\}_{i=1}^{N} = \int yp(y|x, \{(x_i, y_i)\}_{i=1}^{N})dy \tag{46}$$

$$= \int yp(y|x, \theta)p(\theta|\{(x_i, y_i)\}_{i=1}^{N})d\theta dy \tag{47}$$

$$= \int y\mathcal{N}(y; \theta^{\top}xp, \sigma^2_{e})\mathcal{N}(\theta; \mu_{\theta}, \sigma^2_{\theta})d\theta dy \tag{48}$$

$$= \int \theta^{\top}x\mathcal{N}(\theta; \mu_{\theta}, \sigma^2_{\theta})d\theta \tag{49}$$

$$= \mu^{\top}_{\theta}x \tag{50}$$

2.4. Inference beyond supervised learning

3. Regresión No Lineal

El concepto de regresión lineal puede ser extendido para modelar relaciones no lineales entre variables de entrada y salida. Esto manteniendo los supuestos generales del modelo y la simpleza de la solución de las ecuaciones que se desprenden. Para lograr esto se utiliza una familia de transformaciones no lineales.

Considere el conjunto de entrenamiento

$$\{(x_i, y_i)\}_{i=1}^{N}, \quad (x_i, y_i) \in \mathbb{R}^M \times \mathbb{R}^1, \forall i = 1, \ldots, N \tag{51}$$

Adicionalmente considere la función a valores vectoriales

$$\begin{aligned} \Phi \colon \mathbb{R}^M &\to \mathbb{R}^D \\ x &\mapsto \Phi(x) = [\phi_1(x), \ldots, \phi_D(x)] \end{aligned} \tag{52}$$

donde $\{\phi_i\}_{j=1}^{D}$ son funciones escalares.

Observaciones:

1) Nos referiremos a $\Phi(x)$ como *características* y a x simplementes como *datos crudos*.

2) La función $\Phi : x \mapsto \Phi(x)$ puede depender de parámentros, sin embargo, por ahora eso no es importante.

Usando el vector de características como entrada a un modelo lineal, podemos definir el siguiente modelo de regresión:

$$y = \Phi(x)\theta + \epsilon \tag{53}$$

El cual puede ser escrito de forma vectorial de acuerdo a

$$Y = \Phi(X)\theta + \boldsymbol{\epsilon} \tag{54}$$

$$Y = [y_1, \ldots, y_N]^T$$

$$\Phi(X) = \begin{bmatrix} \phi_1(x_1) & \ldots & \phi_D(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1(x_N) & \ldots & \phi_D(x_N) \end{bmatrix} \tag{55}$$

El ajuste, i.e., determinación de parámetros θ es entonces realizado mediante la minimización del siguiente funcional:

$$J = \frac{1}{2}\sum_{i=1}^{N}(y_i - \Phi(x_i)\theta)^2 \tag{56}$$

$$= \frac{1}{2}(Y - \Phi(X)\theta)^T(Y - \Phi(X)\theta) \tag{57}$$

$$= \frac{1}{2}Y^TY - \theta^T\Phi(X)^TY + \frac{1}{2}\theta^T\Phi(X)^T\Phi(X)\theta \tag{58}$$

$$\tag{59}$$

El cual puede ser encontrado derivando J, es decir,

$$\frac{dJ}{d\theta} = \Phi(X)^T\Phi(X)\theta - \Phi(X)^TY = 0 \tag{60}$$

$$\theta_{ML} = \arg\min_{\theta} J(\Phi(X)^T\Phi(X))^{-1}\Phi(X)^TY \tag{61}$$

De forma homologa al caso lineal, el caso regularizado es

$$J_r = \frac{1}{2}\sum_{i=1}^{N}(y_i - \Phi(x_i)\theta)^2 + \rho\left\|\theta\right\|^2 \tag{62}$$

y su solución es

$$\theta = (\Phi(X)^T\Phi(X) + \rho\mathbb{I})^{-1}\Phi(X)^TY \tag{63}$$

Observaciones:
1) Es importante notar que el modelo de regresión no lineal es **lineal en los parámetros**, no así en los datos.
2) Como se mencionó anteriormente el modelo de regresión lineal es una generalización del caso lineal, esto es claro si se considera lo siguiente

$$\begin{aligned}\phi_0(x) = 1, \phi_1(x) = x\\ \Phi = [\phi_0, \phi_1]\\ \theta = [\theta_0, \theta_1]^T\\ Y = \Phi(X)\theta + \epsilon\end{aligned}$$

3) La solución del caso no lineal $\theta_{nl} = (\Phi(X)^T\Phi(X))^{-1}\Phi(X)^TY$ es similar a la solución del caso lineal. Esto puede ser explicado por el hecho que hacer regresión no lineal con datos $\{(x_i, y_i)\}_{i=1}^N$ de $\mathbb{R}$ a $\mathbb{R}$, puede ser visto como una regresión lineal con datos $\{(\Phi(x_i), y_i)\}_{i=1}^N$ de $\mathbb{R}^D$ a $\mathbb{R}$.

3.1. Bases de funciones

La selección de la base de funciones es fundamental para el desempeño del método, si la familia de funciones no captura el comportamiento no lineal de los datos es muy improbable que el ajuste de la curva sea bueno. En general la elección de la base de funciones debe ser evaluada caso a caso.

A continuación se dan algunos ejemplos de bases de funciones que son usualmente utilizadas por su flexibilidad y/o simpleza.

- **Bases Polinomiales:** La base es $\Phi = \{\phi_i\}_{i=0}^D$, donde $\phi_i(x) = x^i$, de esta forma

$$\Phi(X) = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^D \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^D \end{bmatrix} \tag{64}$$

- **Bases Sinusoidales:** La base es $\Phi = \{\phi_i\}_{i=0}^{D}$, donde $\phi_i(x) = \cos(i\frac{2\pi}{2T}x)$, a medida que aumente i mayor es la oscilación de la función, T controla el periodo.

$$\Phi(X) = \begin{bmatrix} 1 & \cos(i\frac{2\pi}{2T}x_1) & \dots & \cos(D\frac{2\pi}{2T}x_1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \cos(1\frac{2\pi}{2T}x_N) & \dots & \cos(D\frac{2\pi}{2T}x_N) \end{bmatrix} \tag{65}$$

- **Escalones** La siguiente base de funciones está compuesta funciones escalón que valen 1 dentro de un conjunto especifico y 0 en cualquier otro punto. Sean $c_1, c_2, \dots, c_D$ una colección de valores crecientes, se define lo siguiente

$$C_0(x) = I_{(-\infty, c_1)}(x) \tag{66}$$
$$C_i(x) = I_{[c_i, c_{i+1})}(x), \quad \forall i = 1, \dots, D-1 \tag{67}$$
$$C_D(x) = I_{[c_D, +\infty)}(x) \tag{68}$$
$$I_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases} \tag{69}$$

De esta forma la base de funciones $\Phi = \{\phi_i\}_{i=0}^{D}$ queda definida por

$$\phi_i(x) = C_i(x), \quad \forall i = 0, \dots, D \tag{70}$$
$$\Phi(X) = \begin{bmatrix} 1 & C_0(x_1) & \dots & C_D(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & C_0(x_N) & \dots & C_D(x_N) \end{bmatrix} \tag{71}$$

- **Polinomios por partes (Splines)**:

Una opción más flexible a hacer un ajuste polinomial en el dominio de los datos, es utilizar polinomios truncados definidos en intervalos, por ejemplo se puede ajustar

$$y = \begin{cases} \beta_{01} + \beta_{11}x + \beta_{21}x^2 + \beta_{31}x^3 + \epsilon, & x < c \\ \beta_{02} + \beta_{12}x + \beta_{22}x^2 + \beta_{32}x^3 + \epsilon, & x \geq c \end{cases} \tag{72}$$

El problema con el modelo es que en el punto c no existe ninguna restricción de continuidad. Para arreglar se considera la siguiente construcción.

Se considera $\xi_1, \dots, \xi_K$ una secuencia de puntos crecientes que serán los puntos en que cambia el régimen del polinomio, estos puntos se conocen como nudos. Ahora se define el polinomio truncado de grado D con un nudo en ξ_k

$$(x - \xi_k)_+^D = \begin{cases} 0, & x < \xi_k \\ (x - \xi_k)^D, & x \geq \xi_k \end{cases} \tag{73}$$

Con esto polinomio truncado de grado D y con K nudos queda definido por

$$y = \beta_0 + \sum_{d=1}^{D} \beta_d x^d + \sum_{k=1}^{K} (x - \xi_k)_+^D \tag{74}$$

$$\Phi(X) = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^D & \dots & (x_1 - \xi_1)_+^D & \dots & (x_1 - \xi_K)_+^D \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^D & \dots & (x_N - \xi_1)_+^D & \dots & (x_N - \xi_K)_+^D \end{bmatrix} \tag{75}$$

3.2. Clasificación Lineal

3.2.1. Caso 2 clases

Dado un conjunto de datos en el plano donde cada punto pertenece a una dos categorías distintas $\mathcal{C}_1$ y $\mathcal{C}_2$, se quiere abordar el problema de clasificación utilizando un modelo lineal, el modelo más simple que se puede plantear es el siguiente:

$$y(x) = \theta^T x + w \tag{76}$$

Este modelo funciona de la siguiente manera, x será asignado a $\mathcal{C}_1$ si $y(x) \geq 0$ y será asignado $\mathcal{C}_2$ en el caso contrario.

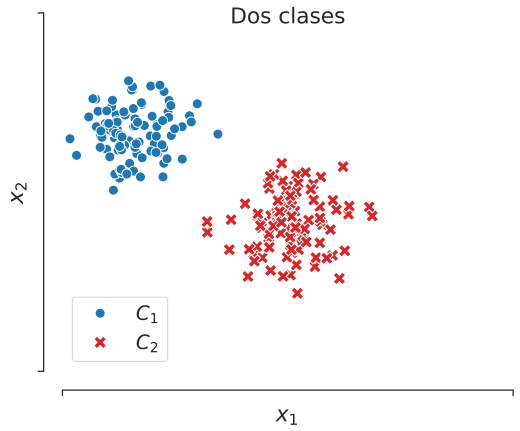


Figura 4: Ejemplo de datos, los puntos en azul pertenecen a $\mathcal{C}_1$, los rojos a $\mathcal{C}_2$.

La superficie de decisión queda definida por $y(x) = 0$, si $x \in \mathbb{R}^D$, entonces la superficie de decisión corresponde a un hiperplano de dimensión $D - 1$.

Sean x_1, x_2 dos puntos en la superficie de decisión, entonces se cumple lo siguiente:

$$\begin{aligned} 0 &= y(x_1) - y(x_2) \\ &= \theta^T x_1 + w - \theta^T x_2 - w \\ &= \theta^T (x_1 - x_2) \end{aligned} \tag{77}$$

Esto muestra que θ es ortogonal a cualquier vector que esté contenido dentro del hiperplano de decisión, esto nos dice que el vector θ controla la inclinación del hiperplano.

Por otro lado, si $y(x) = 0$:

$$\frac{\theta^T x}{\|\theta\|} = -\frac{w}{\|\theta\|} \tag{78}$$

Esto muestra que la distancia normal de un punto x en la superficie de decisión al origen está dada por el parámetro w.

Más aun $y(x)$ da una media con signo sobre cuanto es la distancia perpendicular r entre el punto x y la superficie de decisión. Dado cualquier punto x, este puede ser descompuesto en su proyección ortogonal sobre la superficie de decisión $x_\perp$ y la componente restante respecto al vector θ, de forma que:

$$x = x_\perp + r\frac{\theta}{\|\theta\|} \tag{79}$$

$$y(x) = \theta^T x + w \tag{80}$$

$$= \theta^T x_\perp + r\frac{\theta^T\theta}{\|\theta\|} + w \tag{81}$$

$$\Rightarrow \quad r = \frac{\theta^T + w}{\|\theta\|} = \frac{y(x)}{\|\theta\|} \tag{82}$$

3.2.2. Caso multi clase

Si se tienen K clases $\mathcal{C}_k$ distintas, un enfoque posible para abordar el problema de clasificación es considerar K discriminantes lineales.

$$y_k(x) = \theta_k^T x + w_k, \quad k = 1, \ldots, K \tag{83}$$

Donde x será asignado a la clase $\mathcal{C}_k$ si y solo si $y_k(x) > y_j(x), \quad \forall j \neq k$, es decir:

$$\mathcal{C}(x) = \arg\max_k y_j(x) \tag{84}$$

3.2.3. Clasificación con mínimos cuadrados

Ya planteado el modelo, la pregunta que queda por responder pasa a ser como determinar θ y w, dado los métodos presentados en las clases anteriores el enfoque de mínimos cuadrados parece una respuesta natural a este problema, en primer lugar se introducirá un poco de notación para plantear el problema de una forma cómoda.

Dadas $\{\mathcal{C}_k\}_{k=1}^K$ clases distintas y un punto x usaremos el vector $t \in \{0,1\}^K$ para codificar la pertenencia de x a una clase especifica, por ejemplo: si $x \in \mathcal{C}_j$ entonces el vector t asociado estará compuesto

$$x \in \mathcal{C}_j \Leftrightarrow t_j = 1 \wedge t_i = 0, \quad i \neq j \tag{85}$$

esta codificación también es conocida como 'one-hot'.

Cada clase $\mathcal{C}_k$ posee su propio modelo lineal:

$$y_k(x) = \theta_k^T x + w_k$$

El sistema puede ser reescrito de forma matricial como:

$$y(x) = \tilde{\Theta}^T \tilde{x} \tag{86}$$

Donde la K-esima columna de $\tilde{\Theta}$ es un vector de $D+1$ dimensiones definido por $\tilde{\theta} = (w_k, \theta_k^T)^T$ y $\tilde{x} = (1, x^T)^T$ corresponde al vector x aumentado. De esta forma un punto x será asignado a la clase que tenga mayor $y_k = \tilde{\theta}_k^T \tilde{x}$.

Considerando un conjunto de entrenamiento $\{x_n, t_n\}_{n=1}^N$, definimos las siguientes matrices:

$$T = [t_1, t_2, \ldots, t_N]^T \tag{87}$$

$$\tilde{X} = [\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_N]^T \tag{88}$$

Finalmente para determinar los coeficientes de $\tilde{\Theta}$, minimizamos la suma de los errores cuadráticos de forma similar al problema de regresión, la función de costo puede ser escrita de la siguiente manera:

$$E_D(\tilde{\Theta}) = Tr\{(\tilde{\Theta}\tilde{X} - T)^T(\tilde{\Theta}\tilde{X} - T)\} \tag{89}$$

Donde Tr corresponde al operador traza, al derivar e igualar a 0 la expresión (92), la solución es:

$$\tilde{\Theta} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}T = \tilde{X}^*T \tag{90}$$

Donde $\tilde{X}^*$ denota la pseudo-inversa de la matriz $\tilde{X}$, finalmente la solución es:

$$y(x) = T^T(\tilde{X}^*)\tilde{x} \tag{91}$$

El problema con este enfoque es que mínimos cuadrados es muy sensible a la presencia de puntos aislados, dada la penalización cuadrática, los puntos lejanos al promedio de los datos tiene una influencia mucho mayor, lo que puede empeorar considerablemente los resultados.

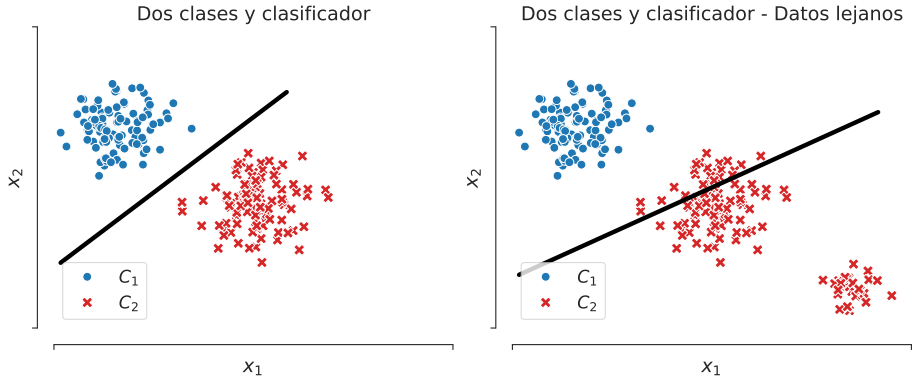


Figura 5: Ejemplo ilustrativo sobre como los puntos lejanos pueden afectar negativamente los resultados.

3.2.4. Discriminante lineal de Fisher

Una forma de ver un modelo de clasificación es en termino de reducción de dimensionalidad. Consideremos el caso en que hay 2 clases y un vector $x \in R^D$ que proyectamos a 1 dimensión de la siguiente manera:

$$y = \theta^T x \tag{92}$$

Si definimos un umbral w, asignamos x a $\mathcal{C}_1$ si $y \geq -w$ y x a $\mathcal{C}_2$ en el caso contrario, recuperamos el modelo lineal que se discutido a través de esta capítulo.

En general al proyectar D dimensiones a 1 se pierde gran cantidad de la información, esto se expresa en el hecho que clases claramente separadas en el espacio D-dimensional pueden traslaparse al ser proyectadas a 1 dimensión. Sin embargo al ajustar las componentes de θ, podemos seleccionar una proyección que maximice la separación entre clases.

Para comenzar sean $N_1 = |\mathcal{C}_1|$ y $N_2 = |\mathcal{C}_2|$, con esto calculamos los promedios por clase.

$$\mathrm{m}_1 = \frac{1}{N_1}\sum_{n\in\mathcal{C}_1} x_n \qquad \mathrm{m}_2 = \frac{1}{N_2}\sum_{n\in\mathcal{C}_2} x_n \tag{93}$$

La medida más simple de separación entre las clases cuando son proyectadas sobre θ es la separación entre los promedios de clase, esto sugiere elegir un θ que maximice la siguiente expresión

$$m_1 - m_2 = \theta^T(\mathrm{m}_1 - \mathrm{m}_2) \tag{94}$$

Donde $m_k = \theta^T \mathrm{m}_k$ corresponde al promedio de la clase $\mathcal{C}_k$ proyectado sobre θ. Esta expresión puede ser arbitrariamente grande si escalamos θ, para evitar imponemos que $\|\theta\|_2 = 1$. Usando multiplicadores de Lagrange para optimizar el problema restringido se llega a que $\theta \propto (\mathrm{m}_1 - \mathrm{m}_2)$. El problema con esta solución es que pueden existir 2 clases bien separadas en el espacio D-dimensional, pero al proyectar los datos sobre la recta que une sus promedios, las proyecciones de cada clase se traslapen.

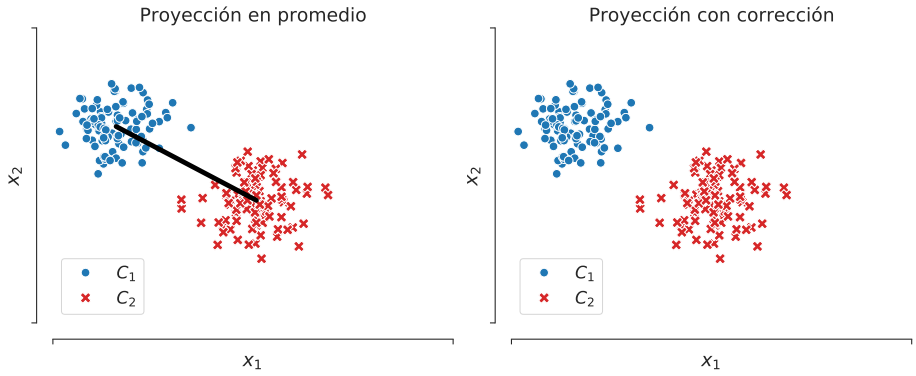


Figura 6: (a) Resultado obtenido al proyectar sobre la recta que une los promedios de cada clase. (b) Resultado al considerar una corrección que considera minimizar la dispersión entre clases.

Para resolver este problema Fisher propuso maximizar una función que de como resultado una gran separación entre clases distintas, pero que al mismo tiempo minimice la separación dentro de una misma clase, de esta forma el traslape entre clases disminuye.

Definimos la varianza dentro una clase como:

$$s_k^2 = \sum_{n \in \mathcal{C}_k} (\theta^T (x_n - \mathrm{m}_k))^2 \tag{95}$$

$$= \sum_{n \in \mathcal{C}_k} (y_n - m_k)^2 \tag{96}$$

Con esto se define la nueva función objetivo de la siguiente manera:

$$J(\theta) = \frac{m_1 - m_2}{s_1^2 + s_2^2} \tag{97}$$

Podemos usar la ecuaciones descritas anteriormente para escribir $J(\theta)$ de forma explicita respecto a θ.

$$J(\theta) = \frac{\theta^T S_B \theta}{\theta^T S_W \theta} \tag{98}$$

Donde S_B es la matriz de covarianza entre clases dada por:

$$S_B = (\mathrm{m}_1 - \mathrm{m}_2)(\mathrm{m}_1 - \mathrm{m}_2)^T \tag{99}$$

S_W es la matriz total de covarianza dentro de clases, dada por:

$$S_W = \sum_{n \in \mathcal{C}_1} (x_n - \mathrm{m}_1)(x_n - \mathrm{m}_1)^T + \sum_{n \in \mathcal{C}_2} (x_n - \mathrm{m}_2)(x_n - \mathrm{m}_2)^T \tag{100}$$

Al derivar $J(\theta)$ se encuentra que la función es maximizada cuando:

$$(\theta^T S_B \theta) S_W \theta = (\theta^T S_W \theta) S_B \theta \tag{101}$$

Por la definición de S_B, vemos que $S_B\theta$ es co-lineal con $(\mathrm{m}_1 - \mathrm{m}_2)$. Más aun no es importante la norma de θ, solo interesa la orientación, por esta razón descartamos los escalares $(\theta^T S_B \theta)$ y $(\theta^T S_W \theta)$, al despejar θ vemos que la solución es:

$$\theta \propto S_W^{-1}(\mathrm{m}_1 - \mathrm{m}_2) \tag{102}$$

3.3. Clasificación No Lineal

3.3.1. El Perceptrón

El perceptrón de Rosenblatt es un modelo de clasificación binario que tuvo mucha importancia en el área de reconocimiento de patrones. El modelo consiste en una función no-lineal fija usada para transformar x en un vector de características $\phi(x)$, que luego es usado para generar un modelo lineal generalizado de la siguiente forma:

$$y(x) = f(\theta^T \phi(x)) \tag{103}$$

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases} \tag{104}$$

Típicamente el vector $\phi(x)$ contiene un intercepto para la recta, es decir $\phi_0 \equiv 1$. El modelo asigna x a la clase $\mathcal{C}_1$ si $y(x) = +1$ y asignará x a la clase $\mathcal{C}_2$ cuando $y(x) = -1$.

Para encontrar el vector θ suele usarse un criterio basado en los datos que fueron clasificados incorrectamente llamado el criterio del perceptrón, este puede deducirse notando que se quiere encontrar un vector de pesos θ tal que para los $x \in \mathcal{C}_1$ cumpla $\theta^T \phi(x) > 0$ y para los $x \in \mathcal{C}_2$ cumpla $\theta^T \phi(x) < 0$, usando el hecho que $y(x) \in \{1, -1\}$, ambas condiciones pueden ser cubiertas por la expresión:

$$\theta^T \phi(x) y(x) > 0, \quad \forall x \in \mathcal{C}_1 \cap \mathcal{C}_2$$

El criterio del perceptrón asocia a los puntos clasificados correctamente 0 error y a los puntos mal clasificados error $-\theta^T \phi(x) y(x)$, de esta forma si denotamos como $\mathcal{M}$ el conjunto de puntos mal clasificados, se debe minimizar la siguiente función objetivo:

$$E_p(\theta) = -\sum_{i \in \mathcal{M}} \theta^T \phi(x_i) y(x_i)$$

Para minimizar esta función se usa descenso de gradiente estocástico, las actualizaciones son de la siguiente manera:

$$\theta^{\tau+1} = \theta^\tau - \eta \nabla E_p(\theta) \tag{105}$$

$$= \theta^\tau + \eta \phi(x_n) y(x_n) \tag{106}$$

La constante τ corresponde a un entero que indexa las iteraciones. Por otra lado η es una constante que se conoce como tasa de aprendizaje, dado que la función del perceptŕon $y(x) = y(x, \theta)$ no cambia si se multiplica θ por una constante cualquiera, sin perdida de generalidad podemos asumir que $\eta = 1$. Es importante notar que al actualizar el vector θ, el conjunto de puntos mal clasificados $\mathcal{M}$ va a cambiar.

La interpretación del algoritmo usado para ajustar el perceptrón es simple, se recorre el conjunto de puntos de entrenamiento $\{x_n\}_{n=1}^N$, si el punto x_n fue clasificado correctamente el vector de pesos de mantiene igual, por otro lado si x_n fue clasificado incorrectamente, el vector θ^τ es actualizado según (89).

3.3.2. Clasificación Probabilista

Los modelos que hemos mencionado anteriormente son de tipo discriminativo, es decir modelan directamente la probabilidad condicional $\mathbb{P}(\mathcal{C}_k|x)$, que da la probabilidad de pertenencia a una clase dado un dato.

Otro paradigma más poderoso a considerar es un enfoque generativo en el cual modelamos $\mathbb{P}(x|\mathcal{C}_k)$ junto con un prior $\mathbb{P}(\mathcal{C}_k)$ para las clases, luego podemos calcular la densidad posterior usando el Teorema de Bayes.

$$\mathbb{P}(\mathcal{C}_k|x) = \frac{\mathbb{P}(x|\mathcal{C}_k)\mathbb{P}(\mathcal{C}_k)}{\mathbb{P}(x)} \tag{107}$$

Considere el siguiente desarrollo, para el caso con 2 clases:

$$\begin{aligned}
\mathbb{P}(\mathcal{C}_1|x) &= \frac{\mathbb{P}(x|\mathcal{C}_1)\mathbb{P}(\mathcal{C}_1)}{\mathbb{P}(x)} && (108)\\
&= \frac{\mathbb{P}(x|\mathcal{C}_1)\mathbb{P}(\mathcal{C}_1)}{\mathbb{P}(x|\mathcal{C}_2)\mathbb{P}(\mathcal{C}_2) + \mathbb{P}(x|\mathcal{C}_1)\mathbb{P}(\mathcal{C}_1)} && (109)\\
&= \frac{1}{1 + \frac{\mathbb{P}(x|\mathcal{C}_1)\mathbb{P}(\mathcal{C}_1)}{\mathbb{P}(x|\mathcal{C}_2)\mathbb{P}(\mathcal{C}_2)}} && (110)\\
&= \frac{1}{1 + \exp(-a)} = \sigma(a) && (111)\\
& && (112)
\end{aligned}$$

Donde $a = \ln(\frac{\mathbb{P}(x|\mathcal{C}_1)\mathbb{P}(\mathcal{C}_1)}{\mathbb{P}(x|\mathcal{C}_2)\mathbb{P}(\mathcal{C}_2)})$.

- La función logística está se define como $\sigma(a) = \frac{1}{1+e^{-a}}$.
- Tiene propiedades útiles como $\sigma(-a) = 1 - \sigma(a)$ y $\frac{d}{da}\sigma(a) = \sigma(a)(1 - \sigma(a))$
- La inversa de la función logística se conoce como logit y es definida por $a(\sigma) = \ln(\frac{\sigma}{1-\sigma})$

Para el caso multi-clase, se puede hacer un desarrollo similar:

$$\begin{aligned}
\mathbb{P}(\mathcal{C}_i|x) &= \frac{\mathbb{P}(x|\mathcal{C}_i)\mathbb{P}(\mathcal{C}_i)}{\sum_{j\neq i}\mathbb{P}(x|\mathcal{C}_j)\mathbb{P}(\mathcal{C}_j)} && (113)\\
&= \frac{\exp(a_i)}{\sum_{j\neq i}\exp(a_j)} && (114)\\
a_i &= \mathbb{P}(x|\mathcal{C}_i)\mathbb{P}(\mathcal{C}_i) && (115)
\end{aligned}$$

La función que aparece se conoce como softmax y corresponde a una generalización de la función logística a más dimensiones, adicionalmente tiene la propiedad de ser una aproximación suave de la función máximo.

-Ejemplo: Normal class-conditional densities

Suponemos que las densidades condicionales de clase distribuyen como una normal multivariada, $p(x|\mathcal{C}_k) \sim \mathcal{N}(\mu_k, \Sigma)$:

$$p(x|\mathcal{C}_k) = \frac{1}{(2\pi)^{\frac{D}{2}}|\Sigma|^{\frac{1}{2}}} \exp(-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)) \tag{116}$$

$$\Rightarrow \quad a = \ln(\exp(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) + \frac{1}{2}(x-\mu_2)^T\Sigma^{-1}(x-\mu_2))) \tag{117}$$

$$= \theta^T x + w \tag{118}$$

Donde:

$$\theta = \Sigma^{-1}(\mu_1 - \mu_2) \tag{119}$$

$$w = \frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 + \mu_2^T\Sigma^{-1}\mu_2) + \ln(\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_1)}) \tag{120}$$

$$\Rightarrow \quad p(\mathcal{C}_k|x) = \sigma(\theta^T x + w) \tag{121}$$

¿Como ajustar los parámetros de las condicionales a la clase y priors respectivamente?

- **Parámetros:**

$$p(\mathcal{C}_1) = \pi \quad \Rightarrow \quad p(\mathcal{C}_2) = 1 - \pi \tag{122}$$

$$p(x|\mathcal{C}_k) = \mathcal{N}(\mu_k, \Sigma); k = 1, 2 \rightarrow \mu_1, \mu_2, \Sigma \tag{123}$$

- **Datos:**

$$T = (t_1, t_2, \ldots, t_N) \in \{0, 1\}^N \tag{124}$$

$$X = (x_1, x_2, \ldots, x_N) \tag{125}$$

- **Likelihood:**

  Se recuerda que $p(x, \mathcal{C}_k) = p(x|\mathcal{C}_k)p(\mathcal{C}_k) = p(\mathcal{C}_k)\mathcal{N}(x|\mu_k, \Sigma)$, se procede a calcular la función de verosimilitud de los datos:

$$p(X, T|\pi, \mu_1, \mu_2, \Sigma) = \prod_{i=1}^{N} p(x_i, t_i|\pi, \mu_1, \mu_2, \Sigma) \tag{126}$$

$$= \prod_{i=1}^{N} p(x_i, \mathcal{C}_1)^{t_i} p(x_i, \mathcal{C}_0)^{1-t_i} \tag{127}$$

$$= \prod_{i=1}^{N} (\pi\mathcal{N}(x_i|\mu_1, \Sigma))^{t_i}((1-\pi)\mathcal{N}(x_i|\mu_2, \Sigma))^{1-t_i} \tag{128}$$

  Más facil que el likelihood, es log-likelihood:

$$\log(L) := \sum_{i=1}^{N} t_i(\log(\pi) + \log(\mathcal{N}(x_i|\mu_1, \Sigma))) + (1-t_i)(\log(1-\pi) + \log(\mathcal{N}(x_i|\mu_2, \Sigma))) \tag{129}$$

1)Derivada con respecto a π:

$$\frac{\partial \log(L)}{\partial \pi} = \sum_{i=1}^{N} \frac{t_i}{\pi} + \frac{1-t_i}{1-\pi} = 0 \tag{130}$$

$$\Rightarrow \quad (i-\pi)\sum_{i=1}^{N} t_i = \pi \sum_{i=1}^{N}(1-t_i) \tag{131}$$

$$\Rightarrow \quad \sum_{i=1}^{N} t_i = \pi N \quad \Rightarrow \quad \pi = \frac{\sum_{i=1}^{N} t_i}{N} = \frac{N_1}{N_1+N_2} \tag{132}$$

Observamos que el valor encontrado para π corresponde al promedio de las clases.

2)Derivada con respecto a μ_1:

$$\frac{\partial \log(L)}{\partial \mu_1} = \sum_{i=1}^{N} t_i \frac{\partial}{\partial \mu_1}(-\frac{1}{2}(x_i-\mu_1)^T\Sigma^{-1}(x_i-\mu_1)) \tag{133}$$

$$= \sum_{i=1}^{N} t_i(\Sigma^{-1}(x_i-\mu_1)) = \Sigma^{-1}\sum_{i=1}^{N} t_i(x_i-\mu_1) = 0 \tag{134}$$

$$\Rightarrow \quad \sum_{i=1}^{N} t_i x_i = \mu_1 \sum_{i=1}^{N} t_i \quad \Rightarrow \quad \mu_1 = \frac{1}{N_1}\sum_{i=1}^{N} t_i x_i \tag{135}$$

De forma análoga:

$$\mu_2 = \frac{1}{N_2}\sum_{i=1}^{N}(1-t_i)x_i \tag{136}$$

3.3.3. Regresión Logística v/s Modelo Generativo

Los supuestos hechos sobre el modelo generativo resultaron en:

$$p(\mathcal{C}_1|x) = \sigma(w^T x) = \frac{1}{1+e^{-w^T x}} \tag{137}$$

Encontrar w de forma directa es mucho más fácil que ajustar todos los parámetros del modelo generativo.

Para dejar este punto en claro, se calculará la verosimilitud de la regresión logística con datos $\{x_i, y_i\}_{i=1}^{N}$, para hacer la notación más compacta denotamos $\sigma_i = \sigma(w^T x_i)$.

$$p(y_{1:N}|x_{1:N}, w) = \prod_{i=1}^{N} p(y_i|x_i, w) \tag{138}$$

$$= \prod_{i=1}^{N} \sigma_i^{y_i}(1-\sigma_i)^{1-y_i} \tag{139}$$

$$NLL = -\log(p(y_{1:N}|x_{1:N}, w)) \tag{140}$$

$$= -\sum_{i=1}^{N} y_i \log(\sigma_i) + (1-y_i)\log(1-\sigma_i) \tag{141}$$

Se procede a calcular el gradiente de NLL respecto a w:

$$\nabla_w NLL = -\sum_{i=1}^{N} y_i \frac{\sigma_i(1-\sigma_i)}{\sigma_i} x_i + (1-y_i)\frac{-\sigma_i(1-\sigma_i)}{\sigma_i} x_i \tag{142}$$

$$= -\sum_{i=1}^{N} y_i(1-\sigma_i)x_i - (1-y_i)\sigma_i x_i \tag{143}$$

$$= \sum_{i=1}^{N} (\sigma_i - y_i)x_i \tag{144}$$

4. Selección de Modelo

Supongamos que tenemos un montón de datos, de los cuales podemos ajustar un millón de modelos, tanto como la imaginación nos limite. Entonces ¿Con cuál te quedarías al final? Cuando nos enfrentamos a la decisión de elegir un modelo usualmente lo que más interesa es la preciso del modelo a la hora de generalizar. El problema de solo buscar precisión es claro: *overfitting* (ver figura 7).

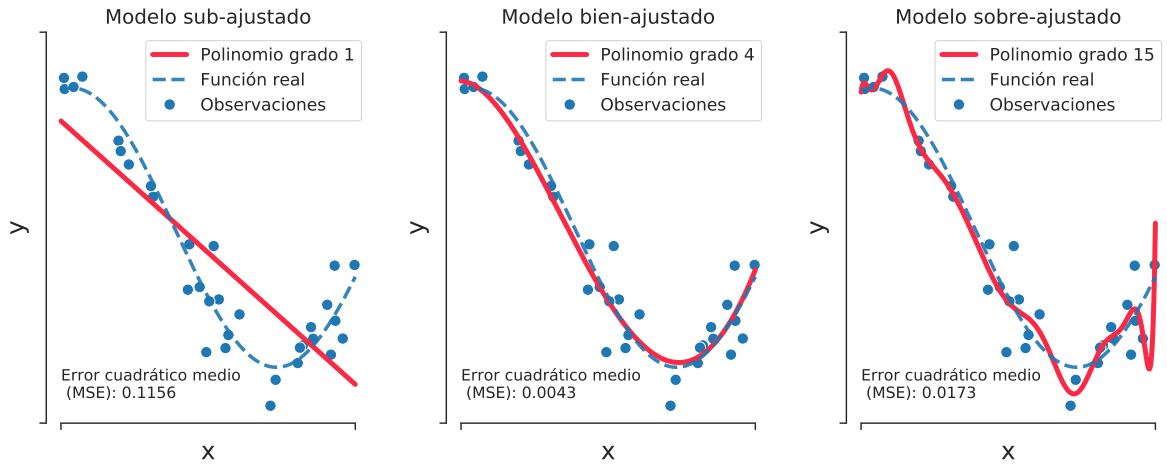


Figura 7: Ejemplos de sub, sobre y correcto ajuste.

La elección de modelo es uno de los problemas más recurrentes en análisis de datos, un ejemplo es la elección de las variables para la regresión ¿Por qué elegir un modelo cuadrático y no uno cúbico? ¿Por qué no incorporar todas las variables cruzadas $(X_1X_2, X_1X_3, ..., X_nX_m)$? Estas preguntas son fácilmente abordables si volvemos a la realidad y queremos correr los modelos en nuestros computadores: Incorporar más variables usualmente aumenta la complejidad computacional y con esto el tiempo de cómputo.

Bajo la filosofía anterior, encontrar un *trade-off* entre *bias* (la flexibilidad del modelo de ajustarse a los datos) y performance sería lo ideal. Es por esta razón que crear un estadístico que pudiera expresar correctamente el compromiso precisión - flexibilidad ayudaría en la elección de modelo. Esto último es lo que busca relizar el *Akaike Information Criterion* (AIC) y el *Bayesian Information Criterion* (BIC) los cuales son dos enfoques distintos para abordar esta problemática.

4.1. Criterio de Información Akaike

La siguiente aproximación es válida cuando $N \to \infty$:

$$-2\mathbb{E}[\log P_{\hat{\theta}}(Y)] \approx \frac{-2}{N}\mathbb{E}[\log L] + 2\frac{d}{N},$$

en donde d es la cantidad de parámetros del modelo y L es la función de verosimilitud definida por:

$$L = \prod_{i=1}^{N} P(y_i|\hat{\theta}). \tag{145}$$

De esta manera se define el estadístico AIC de la forma:

$$AIC(M) = 2d - 2\log(L) \tag{146}$$

Otros autores definen el estadístico como:

$$AIC(M) = 2\frac{d}{N} - 2\frac{log(L)}{N}. \tag{147}$$

Donde N es la cantidad de observaciones.

Ejemplo: Si tenemos un conjunto de modelos $f_\alpha(x)$ indexados por un parámetro α de modo que el modelo α sigue un modelo gaussiano, es decir:

$$\log(L) = -\sum_i (y_i - \hat{f}(x_i))^2/(2\sigma_e^2),$$

entonces el estimador AIC se puede escribir como

$$\text{AIC}(M) = 2d(\alpha) - \overline{err}(\alpha), \tag{148}$$

en donde la función $d(\alpha)$ denota una medida de la complejidad del modelo y $\overline{err}(\alpha)$ denota el error promedio del modelo α.

4.2. Criterio de Información Bayesiano

Nuevamente, si nuestro setting nos permite obtener el máximo de la función de verosimilitud. Podemos utilizar el *Bayesian Information Criterion* (BIC) el cual está definido por:

$$\text{BIC}(M) = log(N)d - 2\log(L) \tag{149}$$

En donde N es la cantidad de observaciones, d es la cantidad de parámetros del modelo y L es el máximo de la función de verosimilitud. Este criterio también es conocido como *Schwarz criterion*. Este estadístico se dice bayesiano por que se obtiene de realizar una aproximación de laplace sobre sobre $\log \mathbb{P}(X|M)$ obteniéndose:

$$\log \mathbb{P}(X|M) \approx \log \mathbb{P}(X|\hat{\theta}_m, M) - \frac{d_m}{2}(\log(N) - \log(2\pi)) \tag{150}$$

Donde $\hat{\theta}_m$ es el máximo del estimador de verosimilitud. Luego para N muy grande se observa que la expresión $-2\log \mathbb{P}(X|M) = BIC(M)$.

Ejemplo: Asumiendo que el modelo sigue un modelo gaussiano, es decir:

$$\log(L) = -\sum_i (y_i - \hat{f}(x_i))^2/(2\sigma_e^2),$$

se tiene que el estadístico BIC está dado por:

$$BIC(M) = \frac{N\overline{err}(M)}{\sigma_e^2} + log(N)d \tag{151}$$

4.3. Evaluación y comparación de modelos

Existen varias formas de evaluar un modelo, una de ella podría ser simplemente evaluar la precisión de sus predicciones. A veces fijarse es natural fijarse en la precisión, como en los problemas de pronóstico. Otras veces la precisión es importante para evaluar diferentes modelos y elegir uno de ellos. En esta sección presentaremos dos maneras distintas de evaluar modelos, cada forma sirve en distintos escenarios, los cuales se discutirán a través de la predicción puntal, que resume la predicción de un conjunto de datos en una solo valor.

4.3.1. Error cuadrático medio

El ajuste del modelo a nuevos datos se puede resumir en una predicción puntual llamada error cuadrático medio, el cual está definido por:

$$MSE(\theta) = \frac{1}{n}\sum_{i=1}^{N}(y_i - \mathbb{E}(y_i|\theta))^2 \tag{152}$$

o su versión ponderada:

$$MSE(\theta) = \frac{1}{n}\sum_{i=1}^{N}\frac{(y_i - \mathbb{E}(y_i|\theta))^2}{\mathbb{V}\text{ar}(y_i|\theta)} \tag{153}$$

4.3.2. log-densidad predictiva o log-verosimilitud

Otra forma de realizar esta evaluación es utilizando el estadístico *log-densidad preditiva* $\log p(y|\theta)$ el cual es proporcional a error cuadrático medio si el modelo es normal con varianza constante. Estudiaremos el caso de un solo punto, para luego extrapolar a más de un punto.

Predictive accuracy para un punto: Sea f el modelo real, y las observaciones (es decir, una realización del dataset y de la distribución $f(y)$), y llamaremos $\tilde{y}$ a la data futura o un dataset alternativos que podemos ver. El ajuste predictivo out-of-sample para un nuevo punto $\tilde{y}_i$ está dado por:

$$\log p_{\text{post}}(\tilde{y}_i) = \log \mathbb{E}[p(\tilde{y}_i|\theta)] = \log \int p(\tilde{y}_i|\theta)p_{\text{post}}(\theta)d\theta \tag{154}$$

Promedio de las distribuciones para un punto: Al tener un dato nuevo $\tilde{y}_i$ entonces se puede calcular el la log-densidad predictiva (elpd, por su sigla en inglés) para el nuevo punto:

$$\begin{aligned}\text{elpd} &= \mathbb{E}_f[\log p_{\text{post}}(\tilde{y}_i)] \\ &= \int \log p_{\text{post}}(\tilde{y}_i) f(\tilde{y}_i) d\tilde{y}\end{aligned} \tag{155}$$

Promedio de las distribuciones para datasets futuros: Como usualmente, no se tiene solo un punto, se debe realizar la suma sobre el conjunto de puntos, calculando así la log-densidad predicitiva puntual (elppd, por su sigla en inglés).

$$\text{elppd} = \sum_{i=1}^{N}\mathbb{E}_f[\log p_{\text{post}}(\tilde{y}_i)] \tag{156}$$

En la práctica, como siempre se tiene la distribución de todos los modelos y la expresión anterior requiere de esto, se suele calcular el estadístico sobre una estimación de un modelo $\hat{\theta}$ (como por ejemplo, el máximo de la función de verosimilitud):

$$\text{elppd}|\hat{\theta} = \sum_{i=1}^{N} \mathbb{E}_f[\log p_{\text{post}}(\tilde{y}_i|\hat{\theta})] \tag{157}$$

Finalmente, una última extensión de este estadístico es cuando se puede tener *draws* de la posterior, es decir, tenemos $\{\theta^s\}_{s=1}^S$, entonces el lppd computado es:

$$\text{computed lppd} = \sum_{i=1}^{N} \left(\frac{1}{S} \sum_{s=1}^{S} p(y_i|\theta^s) \right) \tag{158}$$

4.3.3. Otros métodos

Existen otros estadísticos o métodos que se pueden utilizar para comparar modelos.

- **Deviance Information Criterion:** Se podría decir que es una versión bayesiana de AIC, donde se realizan dos cambios principales 1) se cambia la estimación del estimador de máxima verosimilitud, por el promedio de la posterior y 2) se reemplaza k con una perrción de sesgo de los datos. Formalizando:

  $$\text{DIC} = -2\log(y|\hat{\theta}_{Bayes}) + 2p_{\text{DIC}}, \tag{159}$$

  donde p_{DIC} está dado por:

  $$p_{\text{DIC}} = 2(\log(p(y|\hat{\theta}_{Bayes}) - \mathbb{E}_{\text{post}}(\log(p(y|\theta)). \tag{160}$$

  Nuevamente, puede haber un versión *computada* que está dada por:

  $$\text{computed } p_{\text{DIC}} = 2[\log(p(y|\hat{\theta}_{Bayes}) - \frac{1}{S}\sum_{s=1}^{S} \log(p(y|\theta^s))]. \tag{161}$$

- **Watanabe-Akaike o widely available information criterion:** WAIC es un forma aún más bayesiana para abordar el problema de asginación de evaluación.

  $$\text{WAIC} = -2\text{lppd} + 2p_{\text{WAIC}} \tag{162}$$

  donde hay dos formas de calcular p_{WAIC}.

  $$p_{\text{WAIC1}} = 2\sum_{i=1}^{n} (\log(\mathbb{E}_{\text{post}}[p(y_i|\theta)]) - \mathbb{E}_{\text{post}}[\log(p(y_i|\theta))) \tag{163}$$

  $$p_{\text{WAIC2}} = \sum_{i=1}^{n} (\mathbb{V}\text{ar}_{\text{post}}[\log(p(y_i|\theta))]) \,. \tag{164}$$

- **Leave-one-out cross-validation:** También se puede usar validación cruzada bayesiana para estimar $-2\text{lppd} + 2\text{lppd}_{\text{loo-cv}}$ la cual sirve como estadístico de comparación:

$$\text{lppd}_{\text{loo-cv}} = \sum_{i=1}^{n} \log p_{\text{post}(-i)}(y_i) \tag{165}$$

que se calcula como:

$$\text{computed lppd}_{\text{loo-cv}} = \sum_{i=1}^{n} \log \left(\sum_{s=1}^{S} \frac{1}{S} p(y_i|\theta^{is}) \right) \tag{166}$$

también se puede calcular a versión insesgada, la cual no se utiliza mucho pero se presenta por completitud:

$$\text{lppd}_{\text{cloo-cv}} = \text{lppd}_{\text{loo-cv}} + b. \tag{167}$$

Donde b está dado por:

$$b = \text{lppd}_{-i} - \overline{\text{lppd}_{-i}}, \tag{168}$$

con

$$\overline{\text{lppd}_{-i}} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \log p_{\text{post}(-i)}(y_j) \tag{169}$$

$$\text{computed } \overline{\text{lppd}_{-i}} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \log \left(\frac{1}{S} \sum_{s=1}^{S} p(y_j|\theta^{is}) \right). \tag{170}$$

4.4. Promedio de Modelos

Hay veces que no queremos elegir solo un modelo, puesto que quizás nos interesan las estimaciones de dos o más modelos. Para estos casos, se puede utilizar la técnica de *Model Averaging*, la cual consiste en simplemente ponderar los modelos de la siguiente forma:

$$\hat{\mu} = \sum_{s \in A} c(s) \hat{\mu}_s \tag{171}$$

donde A es el conjunto de todos los modelos. Además, se debe cumplir que:

$$\sum_{s \in A} c(s) = 1 \tag{172}$$

Notemos que la elección:

$$c(s) = \begin{cases} 1 & \text{modelo con mejor criterio.} \\ 0 & \sim \end{cases} \tag{173}$$

es una elección de pesos posible.

4.4.1. Elección de pesos mediante softmax

La principal dificultad para elegir los pesos, está en que se debe asegurar 1) la suma de los pesos sea unitaria, 2) que los pesos sean todos positivos. Una forma de asegurar esto es aplicar una función f positiva sobre una función g de *score*, de modo que:

$$\sum_{s\in A}(f\circ g)(s) > 0 \tag{174}$$

Así los pesos quedan se pueden definir cómo:

$$c_f(s) = \frac{(f\circ g)(s)}{\sum_{z\in A}(f\circ g)(s)} \tag{175}$$

La elección más común de para $f(x) = \text{softmax}(x)$, mientras que para la función de *score* se puede utilizar las recién estudiadas AIC o BIC.

$$c_{AIC}(s) = \frac{\exp\{\frac{1}{2}\text{AIC}_s\}}{\sum_{z\in A}\exp\{\frac{1}{2}\text{AIC}_z\}} \qquad c_{BIC}(s) = \frac{\exp\{\frac{1}{2}\text{BIC}_s\}}{\sum_{z\in A}\exp\{\frac{1}{2}\text{BIC}_z\}} \tag{176}$$

4.4.2. Bayesian Model Averaring

Supongamos que tenemos $\{M_j\}_{j=1}^k$ modelos, de los cuales todos realizan estimaciones razonables de una cantidad μ, dado los datos y. Para realizar el modelo promedio bayesiano de estos modelos es necesario encontrar la distribución posterior de μ dado los datos y sin condicionar a un modelo en específico. De este modo, es necesari:

1. Cada modelo tenga los mismos puntos de interpolación.
2. Prior para cada modelo $p(M_j)$.
3. Prior sobre los parámetros de los modelos $\pi(\theta_i|M_j)$[1].

Luego se puede calcular la función de verosimilitud para los parámetros:

$$\mathcal{L}_{\theta_j}(y) = p(y|\theta_j) \tag{177}$$

Luego se marginaliza sobre los parámetros para obtener la función de verosimilitud del modelo:

$$\lambda_{M_j}(y) = p(y|M_j) \tag{178}$$

$$= \int \mathcal{L}_{\theta_j}(y)\pi(\theta_i|M_j)d\theta_j \tag{179}$$

La expresión (179) es también densidad marginal de las observaciones. Con esto la densidad posterior del modelo es:

$$p(M_j|y) = \frac{p(M_j)\lambda_{M_j}(y)}{\sum_{l=1}^k p(M_l)\lambda_{M_l}(y)} \tag{180}$$

[1]Se puede obtener mediante MCMC.

Finalmente, se puede obtener la distribución posterior para μ asumiendo que el modelo M_j es verdadero ($\pi(\mu|M_j, y)$), y así, la distribución posterior buscada:

$$\pi(\mu|y) = \sum_{j=1}^{k} p(M_j|y)\pi(\mu|M_j, y) \tag{181}$$

5. Redes Neuronales - Editado

5.1. Introducción y Arquitectura

5.1.1. Conceptos Básicos

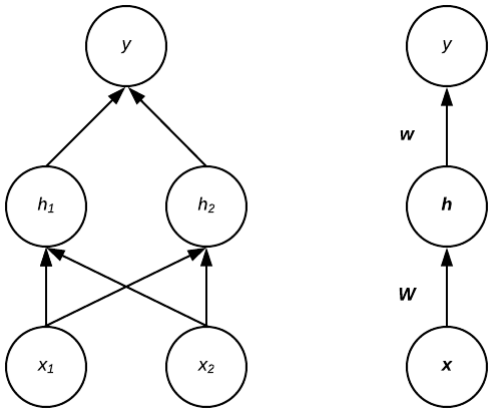
Los modelos escenciales de redes neuronales se conocen como **feedforward neural networks**, o **multilayer perceptrons** (MLPs). Una red neuronal busca aproximar una función f^*. Una red feedforward define un mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ y aprende los parámetros $\boldsymbol{\theta}$ que resultan en la mejor aproximación posible.

Estos modelos se conocen como redes ya que que típicamente son el resultado de composiciones sucesivas de varios tipos de funciones. Por ejemplo, sean $f^{(1)}$, $f^{(2)}$ y $f^{(3)}$ funciones, estas se 'conectan' para formar $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$, se dice que $f^{(1)}$ es la primera **capa** de la red, $f^{(2)}$ la segunda y $f^{(3)}$ la tercera. El largo de esta cadena define la **profundidad** de la red. El uso de redes neuronales con múltiples capas se conoce como **deep learning**, aunque este término se usa para problemas más específicos de aprendizaje de máquinas usando redes neuronales profundas (ver sección 3.5).

Durante el entrenamiento de una red neuronal se busca que $f(\boldsymbol{x}) \approx f^*(\boldsymbol{x})$, donde la data de entrenamiento provee aproximaciones ruidosas de $f^*(\boldsymbol{x})$. Cada dato $\boldsymbol{x}$ viene acompañado de una etiqueta, $y \approx f^*(\boldsymbol{x})$. Los datos de entrenamiento indica qué debe reproducir la red en su última capa, lo cual debe ser un valor aproximado de y. El comportamiento del resto de las capas no tiene una interpretación tan directa como la última capa, por lo que el algoritmo de aprendizaje debe decidir cómo adaptar las capas para que el output producido por la red sea lo más cercano posible a y, la etiqueta, para cada punto $\boldsymbol{x}$. Es por esto que estas capas intermedias se denominan **capas ocultas**.

Estas redes se llaman *neuronales* por su inspiración neurocientífica. Cada capa oculta de la red corresponde a un vector. Se podría pensar que cada elemento de estos vectores tiene un rol similar al de una neurona, en vez de pensar cada capa como una función que produce un mapping de vector a vector, podría considerarse que una capa consiste en muchas **unidades** que actúan en paralelo, cada una representando un mapping de vector a escalar. La elección de las funciones $f^{(i)}(\boldsymbol{x})$, conocidas como **funciones de activación**, en varios casos ha sido guiada por observaciones sobre el comportamiento de neuronas biológicas.

Figura 8: Ejemplo de una red neuronal de 1 capa: (*izquierda*) Se muestra la red con inputs x_1 y x_2, que luego pasan a la capa oculta para producir el output y. (*derecha*) Misma red con una representación vectorial de las capas. La matriz $\boldsymbol{W}$ describe el mapping de $\boldsymbol{x}$ a $\boldsymbol{h}$, y la matriz w el mapping de $\boldsymbol{h}$ a y.



Como se puede apreciar en la figura, los coeficientes $\boldsymbol{W}$ y w se usan para producir el output para la siguiente capa (denominados **pesos**), por lo que la primera operación de esta red será entregar h_1 y h_2 mediante la transformación $\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}^{(1)}$. Luego, esta red aplica $w^T\boldsymbol{h} + b^{(2)}$ para producir el output y. Los coeficientes $\boldsymbol{b}^{(1)}$ y $b^{(2)}$ se conocen como términos de **bias** (sesgo). Todos los parámetros de la red neuronal, los pesos y *bias*, serán agrupados en el término $\boldsymbol{\theta}$.

5.1.2. Función de Costos, Unidades de Output y la Formulación como un Problema Probabilístico

Una de las principales diferencias entre los modelos lineales antes vistos y una red neuronal, es que el uso de ciertas funciones de activación hacen que la función de costos no sea convexa, esto hace que el entrenamiento realizado en base a descenso de gradiente no entregue garantías de que se alcanzará el óptimo global, o una buena solución en términos generales, ya que el algoritmo podría estancarse en un optimo local que entregue resultados pobres.

En la mayoría de los casos, el modelo paramétrico define una distribución $p(y|\boldsymbol{x};\boldsymbol{\theta})$, por lo que los parámetros del modelo se estimarán usando máxima verosimilitud, así, se optimizará la log-verosimilitud negativa, es decir, la **función de costos** a usar será la **cross-entropy**:

$$J(\boldsymbol{\theta}) = -\,\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{\text{data}}}(\log p_{\text{modelo}}(\boldsymbol{y}|\boldsymbol{x})) \tag{182}$$

De esta forma la elección de la **unidad de output** definirá la forma que toma la función de costos, pero no será necesario definir una función especifica para cada problema; en general siempre se resolverá el problema por máxima verosimilitud. La elección en la unidad de output dependerá del tipo de problema que se quiera resolver. Cuando se quiera retornar la media de una distribución Gaussiana condicional, $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}|\hat{\boldsymbol{y}};\boldsymbol{I})$, la unidad de output deberá ser un modelo lineal, $\hat{\boldsymbol{y}} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$, en donde $\boldsymbol{h}$ es el output de la red que proviene de todas las capas ocultas anteriores. En este caso, maximizar la log-verosimilitud es equivalente a minimizar el error cuadrático medio, por esta razón este tipo de output es adecuada para un problema de regresión.

Cuando el problema objetivo es clasificación binaria, una unidad de output apropiada es la función sigmoidal. Para resolver el problema de máxima verosimilitud, se modela utilizando una distribución Bernoulli en y condicional en $\boldsymbol{x}$. Una unidad de output sigmoidal se define como $\hat{y} = \sigma(\boldsymbol{w}^T\boldsymbol{h} + b)$. Esto entrega como output $P(y = 1|\boldsymbol{x})$, por lo que se podrá decidir a qué clase

pertenece y basado en el output de la red, esto se interpreta como la probabilidad de que la observación sea de la clase 1.

Para un problema de clasificación multiclase, la unidad de output apropiada es la generalización de la función sigmoidal, conocida como softmax. El problema es predecir a cuál de n clases pertenece y, por lo que se requiere producir un vector $\hat{\boldsymbol{y}}$, con $\hat{y}_i = P(y = i|\boldsymbol{x})$, por lo que se usará una distribución multinoulli. Primero, una capa lineal predice las log-probabilidades no normalizadas, $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$, y luego se aplica la función softmax para obtener los valores de $\hat{\boldsymbol{y}}$ antes descritos:

$$\log \text{softmax}(\boldsymbol{z})_i = \log \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} = z_i - \log \sum_{j=1}^{n} \exp(z_j) \tag{183}$$

5.1.3. Estructura Interna de la Red

Las capas escondidas proveen a la red neuronal la flexibilidad necesaria para aprender funciones extremadamente complejas, esto mediante el uso de activaciones no lineales. En varios casos estas funciones son incluso no diferenciables, lo que llevaría a pensar de que no son válidas para ser usadas en conjunto con algoritmos que usan descenso de gradiente, pero en la práctica tienen un desempeño suficientemente bueno para ser usadas en tareas de aprendizaje de máquinas. En general, las funciones de activación para las unidades escondidas toman como input un vector $\boldsymbol{x}$ para el cual obtienen una transformación afín $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{b}$, para luego aplicar una transformación no lineal $g(\boldsymbol{z})$. Las unidades escondidas más comunes son las **rectified linear units** o **ReLUs**, $g(\boldsymbol{z}) = \max(0, \boldsymbol{z})$, y sus generalizaciones como **leaky ReLU**, $g(\boldsymbol{z})_i = \max(0, \boldsymbol{z}_i)+\alpha_i \min(0, \boldsymbol{z}_i)$, con α_i una constante pequeña como 0.01; la función sigmoidal $g(\boldsymbol{z}) = \sigma(\boldsymbol{z})$; y la tangente hiperbólica $g(\boldsymbol{z}) = \tanh(\boldsymbol{z}) = 2\sigma(2\boldsymbol{z}) - 1$. A diferencia de las funciones lineales por partes, las unidades sigmoidales se saturan en la mayor parte de su dominio (su gradiente se aproxima a 0) haciendo imposible el aprendizaje por el gradiente cuando esto ocurre, por lo que su uso como unidades escondidas ha sido desalentado.

5.1.4. Diseño de Arquitectura y Teorema de Aproximación Universal

La **arquitectura** de una red neuronal se refiere a la totalidad de su estructura: la cantidad de capas, la cantidad de unidades escondidas, la conexión entre las unidades, etc. Bajo la estructura mostrada para una red feedforward, la primera capa y la segunda capa son de la forma:

$$\begin{aligned} \boldsymbol{h}^{(1)} &= g^{(1)}(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \\ \boldsymbol{h}^{(2)} &= g^{(1)}(\boldsymbol{W}^{(2)T}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \end{aligned} \tag{184}$$

En una arquitectura de este tipo, la principal consideración respecto al diseño es la profundidad y ancho (cantidad de unidades por capa) de cada capa. Una red con solo 1 capa escondida puede ser suficiente para ajustar el set de entrenamiento, cabe destacar que la cantidad de neuronas necesarias en esta única capa podría ser muy grande y la capacidad de generalización puede ser baja, dependiendo de la complejidad de los datos con los que se pretende trabajar. Redes más profundas generalmente permiten disminuir el número de unidades por capa (de esta forma tener una menor cantidad parámetros en total), como así también una mejor capacidad de generalización en el set de testeo, con esto se hace referencia a la capacidad de la red para clasificar correctamente

ejemplos que no fueron usados para el proceso de entrenamiento, naturalmente hacer más intrincado el diseño de la red hará que el proceso de optimización para ajustar los parámetros sea más costoso. La arquitectura ideal se debe encontrar por experimentación con distintas estructuras acompañado de conocimiento sobre el tipo de datos, esto se apoya mediante el monitoreo del error en el set de validación.

Una de las principales justificaciones para usar redes neuronales se debe al **Teorema de Aproximación Universal** (Horniket al., 1989; Cybenko, 1989), el cual muestra que una red feedforward con una capa de output lineal y al menos una capa escondida con función de activación sigmoidal (y otras similares) puede aproximar cualquier función Borel medible [2] de un espacio dimensión finita a otro, con un nivel de error arbitrariamente pequeño, provisto de que hayan suficientes unidades escondidas. Una red neuronal también puede aproximar un mapping de cualquier espacio dimensional finito y discreto a otro, y este teorema también se ha probado para una amplia gama de funciones de activación, como para la más comúnmente usada ReLU (Leshno et al., 1993). La desventaja del teorema es que a pesar que asegura que la red podrá aproximar cualquiera de estas funciones, esto no implica que necesariamente podrá *aprender* la función en sí. Una razón posible es que el algoritmo de optimización podría no encontrar los parámetros de la red que alcancen el nivel de error deseado. Tampoco especifica la cantidad de unidades que son necesarias para alcanzar un nivel de error dado, lo cual podría ser excesivamente grande. Sin embargo en muchos casos modelos más profundos necesitarán menos unidades y potencialmente permiten reducir el error de generalización.

5.2. Entrenamiento de una Red Neuronal

5.2.1. Forward Propagation y Back-Propagation

Al usar una red neuronal feedforward, la información fluye a través de la red desde el ingreso de un input $\boldsymbol{x}$ hasta producir un output $\hat{\boldsymbol{y}}$. Esto se conoce como **forward propagation**. Durante el entrenamiento, *forward propagation* continúa hasta producir el costo escalar $J(\boldsymbol{\theta})$.

El algoritmo de **back-propagation** permite que la información del costo fluya en sentido inverso a través de la red para calcular el gradiente de manera computacionalmente eficiente. El gradiente se calculará de esta forma ya que, aunque es posible obtener una expresión analítica para este, evaluar la expresión puede ser muy caro computacionalmente. Luego de obtener el gradiente, otro algoritmo como descenso de gradiente estocástico (ver sección 4) realiza el aprendizaje usando la expresión que fue calculada. En algoritmos de aprendizaje el gradiente que más comúnmente se requiere obtener es $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, aunque el algoritmo de *back-propagation* no se limita a esto y puede ser usando para otras tareas que involucren obtener derivadas.

Sea $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n, g : \mathbb{R}^m \to \mathbb{R}^n, f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{y} = g(\boldsymbol{x})$ y $z = f(\boldsymbol{y})$, la regla de la cadena indica que:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{185}$$

O de manera equivalente visto en su forma vectorial:

$$\nabla_{\boldsymbol{x}} z = (\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})^T \nabla_{\boldsymbol{y}} z \tag{186}$$

[2] En particular cualquier función continua en un subconjunto cerrado y acotado de $\mathbb{R}^n$ es Borel medible, la clase de funciones Borel medibles es mucho más rica y variada que las funciones continuas.

en donde $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ es la matriz Jacobiana de g, con esto se puede ver que para obtener el gradiente con respecto a la variable $\boldsymbol{x}$, basta multiplicar la matriz Jacobiana por el gradiente con respecto a $\boldsymbol{y}$.

Usualmente se aplicará *back-propagation* a **tensores** de dimensionalidad arbitraria, no solo vectores. Se denota entonces el gradiente de z con respecto a un tensor $\mathbf{X}$ como $\nabla_{\mathbf{X}} z$. Sean $\mathbf{Y} = g(\mathbf{X})$ y $z = f(\mathbf{Y})$, entonces la regla de la cadena se escribe como:

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} \mathrm{Y}_j) \frac{\partial z}{\partial \mathrm{Y}_j} \tag{187}$$

Antes de proseguir a revisar los algoritmos de *forward-propagation* y *back-propagation* se realizará una breve deducción de *back-propagation* que permitirá comprender de mejor manera el proceso de entrenamiento en una red neuronal, en primer lugar de analizará una red neuronal con solo una capa escondida, luego veremos que esto puede ser extendido de forma fácil a arquitecturas más complejas (osea, más capas).

Sea $\{(x_i, y_i)\}_{i=1}^N$ un conjunto de puntos sobre los que se quiere ajustar los pesos de una red neuronal, el conjunto de entrenamiento es tal que $x_i \in \mathbb{R}^p, y_i \in \mathbb{R}^K \quad \forall i = 1, ..., N$, el problema planteado corresponde a uno de clasificación sobre K clases distintas. Adicionalmente se tiene que el error (función de perdida) puede ser escrito como:

$$J(\theta) = \sum_{i=1}^N J_i = \sum_{i=1}^N L(y_i, f(x_i; \theta)) \tag{188}$$

Donde J_i corresponde a la discrepancia entre y_i y el valor entregado por la red $f(x_i; \theta)$ evaluado por la función de perdida L. Para fijar ideas, la arquitectura de la red está compuesta por p unidades de entrada, la capa escondida tiene M neuronas con funciones de activación sigmoides ($\sigma(x) = \frac{1}{1-e^{-t}}$) y la capa de salida (output) tendrá K unidades y una función de salida g. De esta forma el problema puede ser planteado como:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \quad \forall m = 1, ..., M \tag{189}$$
$$N_k = \beta_{0k} + \beta_k^T Z, \quad \forall k = 1, ..., K \tag{190}$$
$$O_k = g_k(N), \quad \forall k = 1, ..., K \tag{191}$$
$$f(X) = O \tag{192}$$
$$\tag{193}$$

Donde $Z = (Z_1, ..., Z_M)$, $N = (N_1, ..., N_K)$, $O = (O_1, ..., O_K)$ y se puede ver que $g : \mathbb{R}^K \to \mathbb{R}^K$, de esta forma los parámetros θ del modelo son:

$$\{\alpha_{0m}, \alpha_m; \quad m = 1, ..., M\} \quad M(p+1) \quad \text{pesos}$$
$$\{\beta_{0k}, \beta_k; \quad k = 1, ..., K\} \quad K(M+1) \quad \text{pesos}$$

De ahora en adelante $J \equiv J(\theta)$, también es importante distinguir que es distinto calcular la derivada de J respecto a un peso que está en la capa de salida y un peso que pertenece a una capa escondida, siendo el primer caso mucho más fácil de calcular, esto se justifica por el hecho

que los valores de la capa de salida participan directamente en el calculo del error J, mientras que los valores de las neuronas escondidas no, esto se hará evidente en los cálculos. Como último preámbulo se definen las siguientes variables que nos ayudaran a hacer los cálculos más claros:

$$\begin{aligned} z_{mi} &= \sigma(\alpha_{0m} + \alpha_m^T x_i) \\ z_i &= (z_{1i}, ..., z_{Mi}) \\ n_{ki} &= \beta_{0k} + \beta_k^T z_i \\ n_i &= (n_{1i}, ..., n_{Ki}) \\ o_i &= g(n_i) \\ o_{ki} &= (o_i)_k \end{aligned}$$

Considerando que derivar es una operación lineal calcularemos la derivada de J_i, a partir de este termino podemos obtener la derivada de J sumando sobre todo $i = 1, ..., N$.

$$\frac{\partial J_i}{\partial \beta_{km}} = \frac{\partial J_i}{\partial o_{ki}} \frac{\partial o_{ki}}{\partial n_{ki}} \frac{\partial n_{ki}}{\partial \beta_{km}} \tag{194}$$

$$= \frac{\partial J_i}{\partial o_{ki}} g_k'(n_{ki}) z_{mi} \tag{195}$$

Dado que o_{ki} pertenece a la capa de salida, esto implica que participa directamente en la expresión de J_i y por lo tanto la derivada $\frac{\partial J_i}{\partial o_{ki}}$ puede ser calculada directamente. El calculo recién hecho se conoce como regla delta ('delta rule') y es el paso principal de *back-propagation*, este consiste en usar la regla de la cadena 2 veces sucesivamente.

Como se mencionó con anterioridad calcular la derivada de J_i respecto a un peso perteneciente a una capa escondida es más engorroso, ya que las neuronas en las capas intermedias no participan directamente en el error, aún así es posible y el calculo entrega una formula recursiva que permite llegar a una expresión cerrada y explicita.

$$\frac{\partial J_i}{\partial \alpha_{ml}} = \frac{\partial J_i}{\partial z_{mi}} \frac{\partial z_{mi}}{\partial \alpha_{ml}} \tag{196}$$

$$= \left[\sum_{k=1}^{K} \frac{\partial J_i}{\partial o_{ki}} \frac{\partial o_{ki}}{\partial n_{ki}} \frac{\partial n_{ki}}{\partial z_{mi}} \right] \frac{\partial z_{mi}}{\partial \alpha_{ml}} \tag{197}$$

$$= \left[\sum_{k=1}^{K} \frac{\partial J_i}{\partial o_{ki}} \frac{\partial o_{ki}}{\partial n_{ki}} \frac{\partial n_{ki}}{\partial z_{mi}} \right] \sigma'(\alpha_0 + \alpha_m x_i) x_{il} \tag{198}$$

$$= \left[\sum_{k=1}^{K} \frac{\partial J_i}{\partial o_{ki}} g'(n_{ki}) \beta_{km} \right] \sigma'(\alpha_0 + \alpha_m x_i) x_{il} \tag{199}$$

$$= \sum_{k=1}^{K} \frac{\partial J_i}{\partial o_{ki}} g'(n_{ki}) \beta_{km} \sigma'(\alpha_0 + \alpha_m x_i) x_{il} \tag{200}$$

Al llegar a la expresión final se puede ver que todos los términos pueden ser calculados de forma explicita, en este punto se puede apreciar porqué el nombre *back-propagation*, a partir de

los cálculos obtenidos en las capas exteriores se pueden obtener derivadas de los pesos en capas anteriores, en este sentido la información se propaga 'hacia atrás' en la red. Finalmente reescribimos (195) y (200) como:

$$\frac{\partial J_i}{\partial \beta_{km}} = \delta_{ki} z_{mi} \qquad \frac{\partial J_i}{\partial \alpha_{ml}} = s_{mi} x_{il} \tag{201}$$

Donde:

$$\delta_{ki} = \frac{\partial J_i}{\partial o_{ki}} g_k'(n_{ki}) \tag{202}$$

$$s_{mi} = \sigma'(\alpha_0 + \alpha_m x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki} \tag{203}$$

Las cantidades δ_{ki} y s_{mi} son errores del modelo en la capa de salida y capas escondidas respectivamente, las 4 ultimas ecuaciones presentadas son conocidas como como ecuaciones de *back-propagation*. Para extender el desarrollo a una red neuronal con más capas es importante notar que a partir de la capa final y la anterior se logró calcular todas las derivadas del gradiente del error, si se considerará la capa de entrada (input) como otra capa escondida más, se pueden repetir los mismos razonamientos y extenderlos a un nivel de profundidad arbitrario.

A continuación se presentan los algoritmos de *forward propagation* y *back-propagation* para una red MLP en donde se conectan todas las unidades de una capa con la siguiente (fully connected MLP).

Algoritmo 1 Forward Propagation

Requerir: Profundidad de la red, l
Requerir: $\boldsymbol{W}^{(i)}, i \in \{1, ..., l\}$, pesos de la red
Requerir: $\boldsymbol{b}^{(i)}, i \in \{1, ..., l\}$, parámetros bias de la red
Requerir: $\boldsymbol{x}$, el input
Requerir: $\boldsymbol{y}$, el output target
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, ..., l$:
$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

Algoritmo 2 Back-Propagation

Luego de completar forward propagation:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, ..., 1$:
$\nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T}$
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$

Con este algoritmo se obtienen los gradientes con respecto a todos los parámetros hasta la primera capa, propagando el gradiente desde una capa a la anterior mediante la última actualización del algoritmo para cada iteración.

5.3. Regularización para una Red Neuronal

Las redes neuronales y algoritmos de deep learning son aplicados a tareas extremadamente complejas como lo son el procesamiento de imágenes, audio, y texto. Controlar la complejidad de un modelo no solo se reduce a encontrar el tamaño y cantidad de parámetros adecuados, como se ha visto para otros modelos de aprendizaje de máquinas, sino que en la práctica el modelo con el mejor ajuste por lo general será un modelo grande (profundo) que ha sido regularizado apropiadamente.

5.3.1. Regularización L^2

Una regularización que se basa en limitar la norma de los parámetros del modelo es la ya conocida **regularización L^2** (o **ridge regression**), mediante la cual se obtiene la función objetivo regularizada $\tilde{J}$:

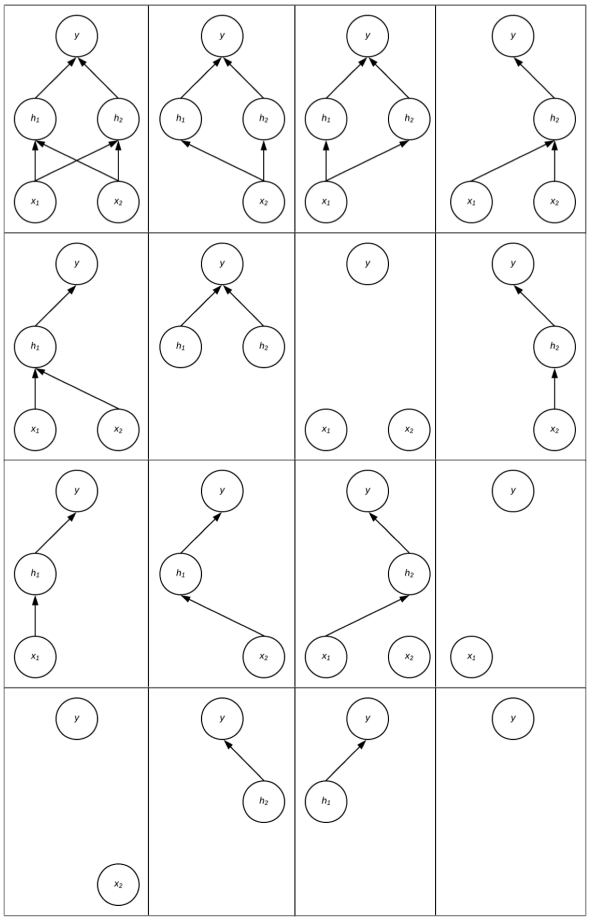
$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \frac{\alpha}{2}||\boldsymbol{\theta}||_2^2 \tag{204}$$

en donde el hiperparámetro $\alpha \in [0, \infty[$ indica que tanta importancia se le da al termino de regularización sobre el objetivo, no habrá regularización cuando $\alpha = 0$ y se observará un mayor efecto regularizador a medida que α crece. Cabe destacar que típicamente en una regularización por la norma solo se regularizan los *pesos*, dejando los términos de *bias* sin regularizar. Esto ya que cada término de *bias* controla el comportamiento de solo 1 variable implicando que no se introduce mucha varianza **[(*overfitting*), no se si es pertinente esta acotación]*** al dejarlos sin regularizar, por otro lado regularizar los *bias* puede inducir un alto nivel de *underfitting*.

5.3.2. Dropout

Bagging consiste en entrenar múltiples modelos y evaluarlos en cada dato del set de testeo. Esto no es práctico para redes neuronales[**Duda, según lo que vi baggin se refiera a cuando se entrenan muchos clasificadores y luego votan sobre el resultado**]*, ya que un solo modelo puede ser muy caro de entrenar y evaluar. **Dropout** provee una aproximación barata (computacionalmente) para entrenar y evaluar *bagged* ensambles compuestos por una cantidad exponencial de redes neuronales. Dropout entrena los ensambles de posiblemente todas las subredes que se puedan formar al remover unidades (que no sean las de output) de un modelo de red neuronal (ver figura). Esto se puede realizar al multiplicar por 0 el output de alguna unidad para la mayoría de los casos. Específicamente, para entrenar con dropout se usa un algoritmo por mini-batches (ver sección 4) para que en cada iteración se entrene una subred aplicando una máscara binaria a todas las capas de input y escondidas. Los hiperparámetros de este método de regularización corresponden al diseño de la máscara binaria, especificando la probabilidad de que se incluya una unidad de input y la probabilidad de que se incluya una unidad escondida. Típicamente se usan los valores 0.8 y 0.5, respectivamente, sin embargo estos deben ser ajustados para controlar el nivel de regularización (mayor valor para las probabilidades implica menor efecto regularizador) para obtener un desempeño óptimo en el set de validación.

Figura 9: Dropout entrena potencialemente todas las subredes que se puedan formar a partir de la red neuronal original (primer recuadro) al apagar el output que producen las distintas unidades



5.3.3. Otros Métodos de Regularización

Otras formas de regularización también buscan introducir alguna fuente de ruido (como en dropout) para que la red neuronal aprenda principalmente los parámetros más importantes, así logrando un bajo error de generalización. Una de estas técnicas es **dataset augmentation**, que consiste en generar nuevos datos de entrenamiento (inyectando ruido en el set de entrenamiento), creando datos $\boldsymbol{x}$ falsos para los cuales se pueda tener una etiqueta y (por ejemplo, una imagen invertida de un gato sigue siendo un gato), o **entrenamiento adversarial**, en donde se perturban ejemplos para fortalecer a la red (por ejemplo, cambiar pixeles de una imagen que generen cambios imperceptibles para un humano pero que pueden afectar fuertemente la capacidad de predicción de un modelo). Otra técnica es **noise injection** en los pesos (Jim et al., 1996; Graves, 2011), lo cual se puede interpretar como una implementación estocástica de inferencia Bayesiana sobre los pesos, debido a que el aprendizaje consideraría que los pesos son inciertos y, por lo tanto, representables mediante una distribución de probabilidad.

También, por supuesto, **early stopping** es una técnica válida para regularizar redes neuronales.

5.4. Algoritmos de Optimización

Como ya se ha comentado, la optimización en redes neuronales busca resolver un problema particular: encontrar los parámetros $\boldsymbol{\theta}$ que disminuyan significativamente $J(\boldsymbol{\theta})$, que depende de alguna medida de desempeño evaluada en la totalidad del set de entrenamiento, luego se evaluá el error en el set de validación para tener una idea del desempeño, finalmente se ven los resultados en el set de testeo.

Esto se reduce a minimizar la esperanza del error sobre la distribución generadora de los datos, p_{data}:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) \tag{205}$$

Reemplazamos la expresión anterior por un problema sustituto, que consiste en escribir la función de costos como un promedio sobre el set de entrenamiento, como se puede observar la diferencia entre las dos expresiones radica en el hecho que se considera que los datos fueron generados por distribuciones de probabilidad distintas (p_{data} en la primera expresión, $\hat{p}_{\text{data}}$ en la segunda):

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim \hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta), y^{(i)}) \tag{206}$$

5.4.1. Descenso del Gradiente Estocástico y por Batches

Los algoritmos de optimización para aprendizaje de máquinas típicamente actualizan los parámetros usando un valor esperado del costo, obtenido a través de un subset de los términos de la función de costos. La propiedad más usada respecto de la función objetivo (1) es sobre el gradiente, este cumple la siguiente expresión:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = -\,\mathbb{E}_{\boldsymbol{x},y\sim \hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{modelo}}(y|\boldsymbol{x}) \tag{207}$$

Calcular esta expresión es computacionalmente caro, ya que requiere evaluar cada ejemplo del set de entrenamiento, por lo que se puede optar por samplear un pequeño número de ejemplos para obtener este valor esperado, calculando el promedio usando solo estos ejemplos. Los algoritmos de optimización que usan el set de entrenamiento completo para actualizar los parámetros en cada iteración se conocen como **métodos de batch** o **determinísticos**. Los algoritmos que usan un solo ejemplo a la vez se conocen como **métodos estocásticos** u **online** (aunque el término *online* se suele usar para describir un entrenamiento con un flujo continuo de nuevos ejemplos). La mayoría de los algoritmos usados pertenecen a una categoría intermedia, estos son los **métodos de minibatch** o **minibatch estocástico**, los cuales usan un subconjunto de tamaño reducido de la totalidad de los ejemplos. Un criterio guía para decidir el número de batches a usar, es que batches más grandes proveen estimadores más precisos del gradiente, en este caso se obtienen retornos menores a uno lineal.

Una motivación importante para usar descenso de gradiente por mini-batches es que sigue el gradiente del costo que considera el error de generalización (205) mientras no se repitan ejemplos. En la práctica las implementaciones de descenso del gradiente por mini-batches desordenan el set de datos una vez y luego pasan por él múltiples veces.

5.4.2. Algoritmos con Momentum

Aunque los métodos de descenso de gradiente estocástico sigue siendo un algoritmo popular, el aprendizaje a veces puede ser lento. Los algoritmos que incorporan momentum fueron diseñados para acelerar el aprendizaje, especialmente en presencia de altas curvaturas, cuando se tienen gradientes pequeños pero consistentes o gradientes ruidosos. El algoritmo **descenso de gradiente estocástico con momentum** acumula un decaimiento exponencial de media móvil de los gradientes pasados y continúa su movimiento en esta dirección. Un hiperparámetro α determina qué tan rápido las contribuciones de gradientes pasados decaen exponencialmente. Se actualiza mediante:

$$\boldsymbol{v} \longleftarrow \alpha\boldsymbol{v} - \epsilon\nabla_{\boldsymbol{\theta}}\Big(\frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})\Big)$$
$$\theta \longleftarrow \theta + \boldsymbol{v}$$

Con ϵ el learning rate y $\boldsymbol{v}$ la velocidad o momentum.

5.4.3. Algoritmos con Learning Rates Adaptativos

En la práctica el learning rate resulta ser uno de los hiperparámetros más difíciles de ajustar debido a su importante efecto en el desempeño del modelo. La función de costos suele ser altamente sensible (a crecer o decrecer) en algunas direcciones en el espacio de los parámetros e insensible en otras, por lo que hace sentido usar un learning rate distinto para cada parámetro y automáticamente adaptar este parámetro durante el aprendizaje. El algoritmo **AdaGrad** adapta el learning rate de todos los parámetros al escalarlos de manera inversamente proporcional a la raíz cuadrada de la suma de todos las raíces cuadradas históricas del gradiente. Los parámetros con derivadas parciales más grandes tienen un rápido decrecimiento en su learning rate, mientras que los parámetros con derivadas parciales pequeñas decrecen en menor cantidad su learning rate. El efecto neto es mayor progreso en zonas más planas del espacio de los parámetros. Se actualiza mediante:

$$\delta = 10^{-7}; \boldsymbol{r} = 0$$
$$\boldsymbol{g} \longleftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$
$$\boldsymbol{r} \longleftarrow \boldsymbol{r} + \boldsymbol{g}\odot\boldsymbol{g}$$
$$\triangle\boldsymbol{\theta} \longleftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}}\odot\boldsymbol{g}$$
$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \triangle\boldsymbol{\theta}$$

Con δ una constante pequeña para estabilidad numérica (puede ser otra), $\boldsymbol{r}$ la variable de acumulación del gradiente, $\boldsymbol{g}$ el gradiente para el batch, y $\triangle\boldsymbol{\theta}$ la actualización de los parámetros al final de una iteración.

Otras generalizaciones populares son el algoritmo **RMSProp**, que modifica el algoritmo AdaGrad para tener un mejor desempeño en funciones no convexas al cambiar la acumulación del

gradiente por una media móvil que decae exponencialmente, y el algoritmo **Adam**, que combina RMSProp con momentum (con algunas distinciones importantes).

Hasta el momento no hay un algoritmo que tenga un desempeño superior al de los demás en distintos escenarios (Schaul et al., 2014), por lo que se recomienda usar el algoritmo de optimización con el que el usuario se sienta más cómodo al momento de ajustar los hiperparámetros.

5.5. Deep Learning y Otros Tipos de Redes Neuronales

El término **deep learning** se asocia a resolver problemas más intuitivos (y fáciles) para los humanos que hasta hace solo algunos años eran extremadamente difíciles para una máquina, como lo son el reconocimiento de objetos en imágenes (visión de computadores), la traducción de texto desde un lenguaje a otro (machine translation), reconocimiento de voz, entre otros. Los problemas más difíciles para los humanos ya se han estado resolviendo hace mucho tiempo antes del deep learning, como divisar una estrategia ganadora en ajedrez (`https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov`), aunque las redes neuronales profundas han seguido progresando en resolver este tipo de problemas (`https://deepmind.com/research/alphago/`). Las arquitecturas principales que han permitido resolver estos problemas en los últimos años (sumado a los avances en poder de computación y cantidad de datos que existen hoy) se presentan en esta sección: las **redes convolucionales** para procesamiento de imágenes, y las **redes recurrentes** para modelar series de tiempo (e.g., texto, audio). También se presentan otras arquitecturas que son tema activo de investigación en deep learning.: los **autoencoders** y las **redes generativas adversariales**.

5.5.1. Redes Neuronales Convolucionales

Las **redes neuronales convolucionales** (o **CNNs**) son un tipo de redes neuronales que fueron diseñadas para procesar datos con una tipología tipo-*grid* (grilla). Una serie de tiempo que tiene observaciones en intervalos regulares de tiempo se puede pensar como un *grid* de 1 dimensión. Las imágenes se pueden pensar como *grids* de 2-D de pixeles. El nombre de esta arquitectura hace referencia a que usan una operación matemática conocida como convolución.

La operación de **convolución** se define como:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{208}$$

En el contexto de CNNs, el primer argumento a convolucionar, x, es el **input**, y el segundo argumento, w, se conoce como el **kernel**. El output, $s(t)$ se conoce como **feature map**. En aplicaciones de aprendizaje de máquinas, el input serán un arreglo multidimensional de datos, y el kernel un arreglo multidimensional de parámetros que se buscarán aprender. Usualmente, al trabajar con datos en un computador el tiempo se considerará discreto, por lo que resulta conveniente definir la operación de convolución discreta:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{209}$$

Se asumirá que las funciones son 0 en todo su dominio excepto en el set finito de puntos para el cual se guardan valores, permitiendo realizar estas sumatorias infinitas. Las librerías de redes neuronales implementan la función **cross-correlation** y la llaman convolución. Para una imagen I de 2 dimensiones y un kernel K de 2 dimensiones, esto es:

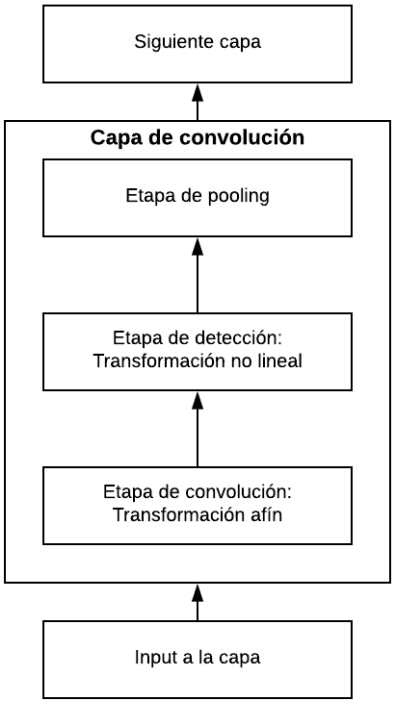
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{210}$$

Esto es equivalente a la operación de convolución discreta con la diferencia que el operador no es conmutativo, $S(i,j) = (I*K)(i,j) \neq (K*I)(i,j)$, lo cual no es importante para implementaciones de redes neuronales.

La motivación por usar convoluciones surge de 3 importantes propiedades: interacciones *sparse*, *parameter sharing*, y representaciones equivariantes. **Interacciones sparse** se refiere a que hay menos conexiones que en una red *fully connected*, esto mediante el uso de kernels de menor dimensionalidad que el input, lo cual implica una eficiencia en términos de memoria por tener que almacenar menos parámetros, y eficiencia computacional por realizar menos operaciones. **Parameter sharing** se refiere a usar los mismo parámetros para distintas funciones dentro del modelo. Esto implica usar los pesos aprendidos en múltiples partes del input (como para reconocer bordes, por ejemplo). Que una función sea **equivariante** significa que si el input cambia, el output cambia de la misma forma. Esto permite que en imágenes la convolución cree un map 2-D dónde ciertos atributos aparecen en el input.

Una capa típica de una red convolucional consta de 3 etapas: primero, aplicar varias convoluciones para producir un set de activaciones lineales. Luego, aplicar una activación no lineal (**detector stage**). Finalmente, una función de *pooling* para modificar aún más el output de la capa (ver figura). Una función de **pooling** reemplaza el output de la red en alguna locación por estadísticos de los outputs cercanos.

Figura 10: Capa de una red convolucional



Max pooling retorna el valor máximo de un output en una vecindad rectangular. Las operaciones de pooling permiten que la red sea invariante a pequeñas transformaciones en el input.

Pooling también es escencial para procesar inputs de tamaño variable (por ejemplo imágenes de distinto tamaño).

Otras diferencias con respecto a la operación de convolución en el contexto de redes neuronales son, por ejemplo, el aplicar múltiples convoluciones en paralelo, esto permite extraer distintos tipos de atributos en vez de 1 solo. Por otro lado, el **stride** hace referencia a cada cuántos pixeles se quieren convolucionar en cada dirección en el output. En la figura se muestra el ejemplo de una convolución con stride. Esta operación permite reducir nuevamente el costo computacional. Esto también implica que el output disminuye su tamaño en cada capa. El uso de *padding* puede revertir esto. **Padding** se refiere a agrandar el input con ceros para hacerlo más amplio. Una convolución en la que no se usa **zero-padding** se conoce como **valid**. Una convolución que mantiene el tamaño desde el input al output se conoce como **same** (ver figura). En la práctica, las capas de una red convolucional usan operaciones entre una convolución valid y same.

Figura 11: Convolución con un stride igual a 2

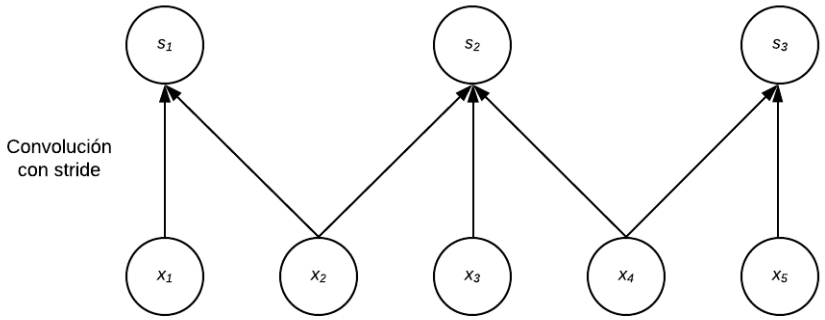
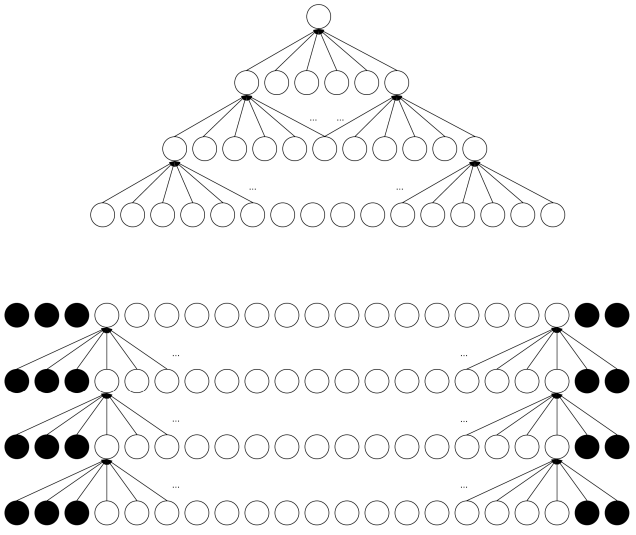


Figura 12: Efecto de no usar zero-padding en una red convolucional (Arriba) y efecto de usar zero padding en una red convolucional (Abajo) en cuanto al tamaño de la red



5.5.2. Redes Neuronales Recurrentes

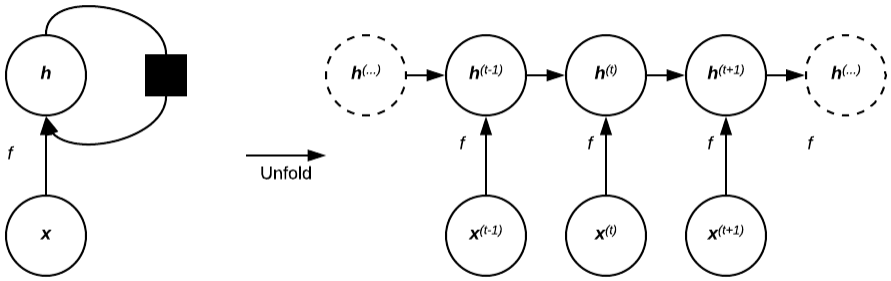
Las **redes neuronales recurrentes** o **RNNs** son una familia modelos de redes neuronales especializados para procesar datos secuenciales, $\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}$. Las RNNs también comparten parámetros, pero en una forma muy distinta que las CNNs. En una RNN, cada miembro del output en una etapa es una función de cada miembro del output de la etapa anterior.

Se denota por $\boldsymbol{h}^{(t)}$ al estado de un sistema dinámico que involucra una recurrencia conducido por un input externo $\boldsymbol{x}^{(t)}$:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}, \boldsymbol{\theta}) \tag{211}$$

La figura siguiente muestra una red recurrente que procesa un input $\boldsymbol{x}$ incorporándolo al estado $\boldsymbol{h}$ que es traspasado a través del tiempo.

Figura 13: Ejemplo de una red recurrente sin output



Las redes recurrentes se pueden construir de muchas formas distintas. Al igual que una red neuronal puede representar casi cualquier función, una red recurrente modela cualquier función que involucre una recurrencia. Se puede representar el estado de una red recurrente luego de t pasos mediante una función $g^{(t)}$:

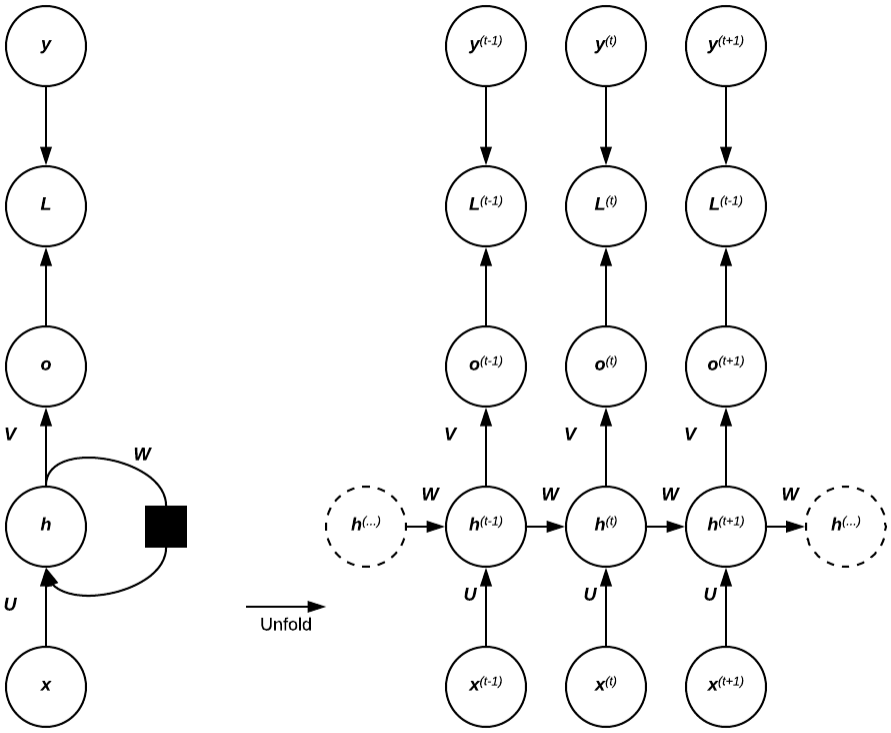
$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, ..., \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}, \boldsymbol{\theta}) \tag{212}$$

Existen varios tipos de RNNs que se han diseñado para distintos fines. Algunos ejemplos de estas son:

- Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre todas las unidades escondidas
- Redes recurrentes que producen un output en cada instante de tiempo y tienen conexiones entre el output a la unidad escondida del siguiente instante
- Redes recurrentes con conexiones entre las unidad escondidas, que procesan una secuencia entera antes de producir el output

La figura muestra un ejemplo de la arquitectura para el primer caso.

Figura 14: Ejemplo de una red recurrente que produce un output en cada instante de tiempo, y que comparte el estado a través del tiempo



Esta red presentada puede ser usada para producir palabras en cada instante, y así producir oraciones que hagan sentido en una conversación. Como ejemplo de entrenamiento de una red con esta arquitectura, en donde en la última capa se decide mediante una función softmax la palabra más probable que deba seguir a la palabra anterior, se tienen las ecuaciones de *forward propagation* para cada instante de tiempo:

$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t-1)} \\ \boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \text{softmax}(\boldsymbol{o}^{(t)}) \end{aligned} \tag{213}$$

El algoritmo aplicado para obtener el gradiente en este tipo de arquitectura se conoce como **back-propagation through time**, y consiste en aplicar el algoritmo de *back-propagation* generalizado para el grafo computacional *unfolded* de la red, como los mostrados en las figuras de redes recurrentes.

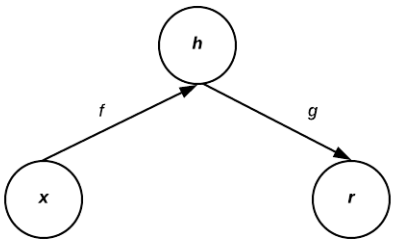
Las redes recurrentes sufren de no poder recordar largas dependencias a través del tiempo, debido a que las recurrencias implican multiplicar una matriz de pesos múltiples veces a través de la red, provocando superficies planas o muy empinadas que resultan en que los algoritmos de aprendizaje por gradiente tengan problemas de **vanishing gradients** o **exploding gradients**, respectivamente. Arquitecturas que han logrado superar esto son las que incluyen compuertas

(funciones sigmoidales) que deciden automáticamente qué olvidar y qué seguir propagando a través de la red. Estos son los modelos de **gated recurrent units** (**GRUs**) y **long-short term memory network** (**LSTM**).

5.5.3. Autoencoders

Un **autoencoder** es una red neuronal que busca replicar el input hacia el output, osea, se busca que la información que entra a la red sea lo más parecida posible a la de salida, para lo cual cuenta con una capa interna $\boldsymbol{h} = f(\boldsymbol{x})$ que codifica el input (genera una representación de este) llamada **encoder** y una función que produce la reconstrucción $\boldsymbol{r} = g(\boldsymbol{h})$, el **decoder**. Un *autoencoder* buscará aprender los *encoder* y *decoder* tales que $g(f(\boldsymbol{x})) = \boldsymbol{x}$ para todo $\boldsymbol{x}$. Como el modelo está forzado a aprender los atributos más importantes para que pueda efectivamente reproducir el input en su output, este aprenderá en general propiedades útiles de los datos de entrenamiento. Los *autoencoders* modernos modelan mappings estocásticos $p_{\text{encoder}}(\boldsymbol{h}|\boldsymbol{x})$ y $p_{\text{decoder}}(\boldsymbol{x}|\boldsymbol{h})$, en vez de funciones determinísticas. En la figura se muestra la arquitectura de un *autoencoder*.

Figura 15: Estructura de un *autoencoder* típico



Una manera de obtener atributos útiles de $\boldsymbol{x}$ es forzando a que el *encoder* tenga una dimensionalidad menor que el input. Este tipo de *autoencoders* se denominan **undercomplete**. El aprendizaje se describe mediante la optimización de la función de pérdida, $L(\boldsymbol{x}, g(f(\boldsymbol{x})))$. Cuando el *decoder* es lineal y la función de pérdida es el error cuadrático medio, un *undercomplete autoencoder* aprende a generar el mismo subespacio que el algoritmo **principal component analysis** (**PCA**), es decir, el *autoencoder* que fue entrenado para reproducir los datos de entrenamiento mediante una reducción de dimensionalidad y una reconstrucción aprendió como efecto colateral el subespacio principal. Es así como entonces, *autoencoders* con *encoders* y *decoders* no lineales pueden aprender representaciones no lineales más poderosas que PCA.

Un *autoencoder* con dimensión de su *encoder* igual a la del input se conoce como **overcomplete**. Estos *autoencoders*, al igual que los *undercomplete*, pueden fallar en aprender una representación útil del input si tienen mucha capacidad, por lo que será importante también regularizar estas redes neuronales.

Otras aplicaciones de los autoencoders, aparte de aprender una reducción de dimensionalidad, es aprender representaciones útiles que sirvan para un posterior modelo de redes neuronales (o, más general, de aprendizaje de máquinas). Por ejemplo, en vez de usar **one-hot-vectors** para representar palabras (en donde se tiene un vector del largo de cierto vocabulario compuesto por ceros excepto para la palabra que se quiere representar, indicando un valor de 1 en esa posición), se pueden usar **embeddings**, que son representaciones del input a un espacio de valores reales. A diferencia de los *one-hot-vectors*, en un *embedding* la distancia entre las representaciones del texto

sí tiene un significado, y este tipo de representaciones podría entregar mejores resultados en la tarea en que se esté usando.

5.5.4. Redes Generativas Adversariales

Una **red generativa adversarial** (o **GAN**) se basa en un escenario de teoría de juegos, en donde una **red generadora** debe competir con un adversario. La red generadora produce muestras $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)})$, mientras que una **red discriminadora** trata de distinguir entre muestras obtenidas de los datos de entrenamiento y muestras generadas por la red generadora. El discriminador retorna una probabilidad, $d(\boldsymbol{x}; \boldsymbol{\theta}^{(d)})$, indicando la probabilidad de que $\boldsymbol{x}$ sea un dato real y no uno simulado.

Para formuuar el aprendizaje, se describe un juego de suma cero en donde una función $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ determina el pago del discriminador, y el generador recibe $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ como pago. Así, durante el entrenamiento cada jugador intenta maximizar su propio pago, para que en convergencia se tenga

$$g^* = \text{argmin}_g \text{max}_d v(g, d) \tag{214}$$

Esto motiva a que el discriminador aprenda a clasificar correctamente entre muestras reales y falsas y, simulatáneamente, el generador intenta engañar al clasificador para que crea que las muestras generadas son reales. En convergencia, las muestras del generador son indistinguibles de los datos reales. Una motivación del uso de GANs es que cuando $\text{max}_d v(g, d)$ es convexa en $\boldsymbol{\theta}^{(g)}$, el procedimiento asegura la convergencia.

Figura 16: Imágenes generadas por una GAN entrenada con el set de datos LSUN. (Izquierda) Imágenes de dormitorios generadas por el modelo DCGAN (imagen de Radford et al., 2015). (Derecha) Imágenes de iglesias generadas por el modelo LAPGAN (imagen de Denton et al., 2015)

6. Support Vector Machines

6.1. Introducción

En este capítulo del apunte, discutiremos las llamadas **máquinas de soporte vectorial** (SVM) que constituyen un set de herramientas para resolver el problema de clasificación de datos. Fue desarrollada en los 90 dentro de la comunidad de computer science y ha crecido enormemente desde entonces. Esta técnica ha demostrado ser útil en diversos escenarios, y es considerado uno de los mejores clasificadores para "llegar y usar".

6.2. Idea general

Los SVM son una generalización de un clasificador más simple e intuitivo llamado el clasificador de márgen máximo (*maximum margin classifier*)

Para estudiar dicho clasficador, comenzaremos estudiando el caso en que tenemos que clasificar los datos en dos grupos, y además supondremos que los datos son linealmente separables (es decir, existe un hiperplano que los separa, ver fig. 17). Esto último es claramente un supuesto muy fuerte, pero lo generalizaremos mas adelante.

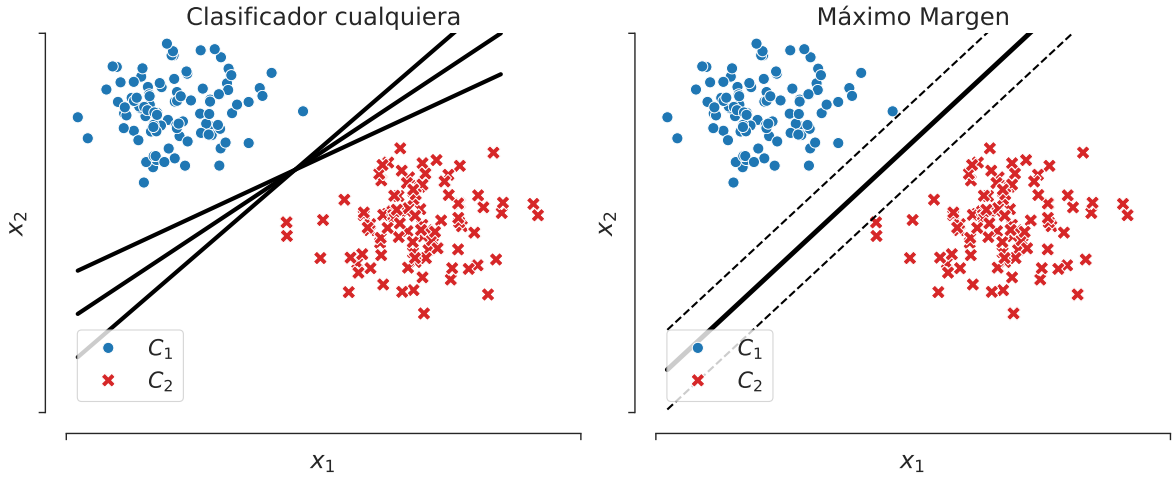


Figura 17: Modelo de máximo margen entre los datos

Como se puede apreciar en la fig. 17 (centro), dados puntos linealmente separables, existen diferentes hiperplanos que separan los datos. Cada una de estas líneas define un modelo, ya que dado un nuevo dato x^*, si este está por encima de la línea, entonces es un dato verde, en cambio si esta por debajo, entonces es un dato rojo.

¿Cuál elegir?

Estudiaremos una respuesta natural al problema: ¿cómo elegir el hiperplano que nos entregue el **mayor margen** posible entre ambos grupos? Esto se traduce en reemplazar el problema de separar mediante una línea, en separar mediante una *cinta* de ancho máximo. A este modelo se le llama el separador de máximo margen que mencionamos antes.

El argumento de ocupar dicho modelo es que uno esperaría que si un modelo tiene un buen margen en los datos de entrenamiento, entonces el **error de generalización** (Cuanto se equivoca el modelo con datos nuevos) debiese ser bajo. Existe un argumento matemático, y a grandes rasgos es que a mayor margen y a medida aumentan los datos, podemos acotar la probabilidad de error (Al estilo desigualdad de Markov) sin embargo los detalles se escapan de los contenidos del curso.

Continuando, en la fig. 17 (derecha) podemos visualizar el modelo. Notar que el margen esta definido solamente por algunos puntos. En la figura son dos, podrían ser más, pero nunca menos por la naturaleza " simétrica " del modelo.

Dichos puntos que definen el márgen son llamados los vectores de soporte (support vectors) que le entregan el nombre al método.

Un ejemplo sencillo para recordar que es lo que hace SVM es imaginar que el ejemplo de los gráficos trata sobre clasificar manzanas y naranjas:
A diferencia de otros métodos que aprenden analizando todos los datos y llegando a una respuesta (Cómo Naive Bayes, Regresión Lineal, etc...) de lo que en promedio podría ser una manzana, SVM mira la manzana mas naranjezca y la naranja mas manzanezca y con esos datos define un hiperplano separador.

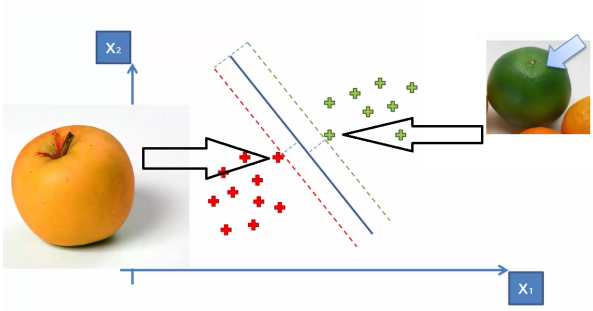


Figura 18: Los vectores de soporte son la manzana más naranjezca y la naranja más manzanezca

6.3. Problema

Como ya mencionamos tenemos datos de entrenamiento $\{x_i\}_{1,\dots,N}$ que suponemos linealmente separables y deseamos encontrar un hiperplano de máximo margen que los separe.
Un hiperplano en general son los $x \in \mathbb{R}^n$ que cumplan al ecuación

$$w \cdot x + b = 0 \tag{215}$$

Donde $w \in \mathbb{R}^n$ es el vector perpendicular al hiperplano y $b \in \mathbb{R}$ es un parámetro. La idea es encontrar w y b tales que entreguen el hiperplano separador de mayor margen. Este problema no tiene solución única (Pues si w es solución entonces λw también). Para evitar ese problema, escalamos el plano de manera que los vectores de soporte (x_+ y x_-) sean tales que $w \cdot x_+ + b = 1$ y $w \cdot x_- + b = -1$ (Pueden haber mas vectores de soporte, pero para efectos del cálculo usaremos dos). Si bien aún no tenemos los vectores de soporte para hacer el escalamiento, este será parte de las restricciones del problema de optimización que resolveremos.

El márgen del hiperplano es la mitad de la diferencia entre ambos vectores de soporte, proyectada en la dirección $\frac{w}{||w||}$ (Ver figura 5).

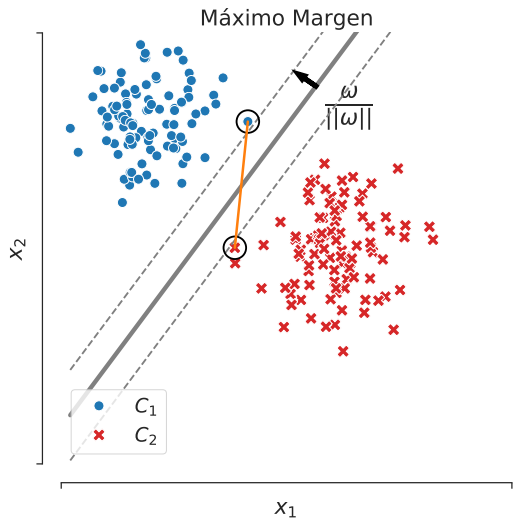


Figura 19: En azul los vectores x_+ y x_-. En rojo el vector $(x_+ - x_-)$. En naranjo la componente del vector rojo, proyectada en la dirección definda por $\frac{w}{||w||}$. Notar que corresponde justamente al doble del margen

Calculamos:

$$\begin{aligned}\frac{1}{2||w||}[w \cdot (x_+ - x_-)] &= \frac{1}{2||w||}[(w \cdot x_+) - (w \cdot x_-)] \\ &= \frac{1}{2||w||}[1 - (-1)] \\ &= \frac{1}{||w||}\end{aligned}$$

Definiendo a y_i como 1 si $w \cdot x_i + b \geq 1$ y $y_i = -1$ cuando $w \cdot x_i + b \leq -1$, podemos formular el problema del hiperplano de margén máximo como

$$\begin{aligned}\max_{w,b} \quad & \frac{1}{||w||} \\ \text{s.a} \quad & y_i(w \cdot x_i + b) \geq 1, \ i = 1, \ldots, N.\end{aligned}$$

Para evitar problemas de diferenciablidad, en realidad se ocupa la siguiente formulazión equivalente

$$\begin{aligned}\min_{w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.a} \quad & y_i(w \cdot x_i + b) \geq 1, \ i = 1, \ldots, N.\end{aligned}$$

Ahora estudiaremos el lagrangiano del problema (Y en consecuencia su dual). Este problema tiene una mejor estructura para optimizar, además que nos servirá para generalizar al caso no lineal.

El lagrangiano es

$$L(w,b,\alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{N} \alpha_i (y_i(w \cdot x_i + b) - 1)$$

Las condiciones de primer orden nos dicen que

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{N} \alpha_i y_i x_i \tag{216}$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_{i=1}^{N} \alpha_i y_i = 0 \tag{217}$$

Reemplazando (2) y (3) en $L(w,b,\alpha)$, maximizando (por ser dual) e imponiendo holgura complementaria, el problema queda

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ \text{s.a} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & \alpha_i \geq 0 \end{aligned}$$

Este problema es de la forma QP (Quadratic Programming) y existen diferentes métodos para resolverlo de manera óptima. Notar la participación del producto punto $\langle \cdot, \cdot \rangle$ en este problema, pues este será el punto de partida del SVM no lineal.

Una vez resuelto el dual, la predicción de un nuevo punto x^* es de la forma

$$\hat{y}(x^*) = \text{sgn}\left(\left[\sum_{i=1}^{N} \alpha_i y_i \langle x_i, x^* \rangle \right] - b \right)$$

Por holgura complementaria tenemos que si x_i no está en el márgen, entonces $\alpha_i = 0$ de modo que x_i no aporta en la predicción $\hat{y}$. Esta propiedad es la que mencionabamos al principio: La predicción solo depende de los vectores de soporte.

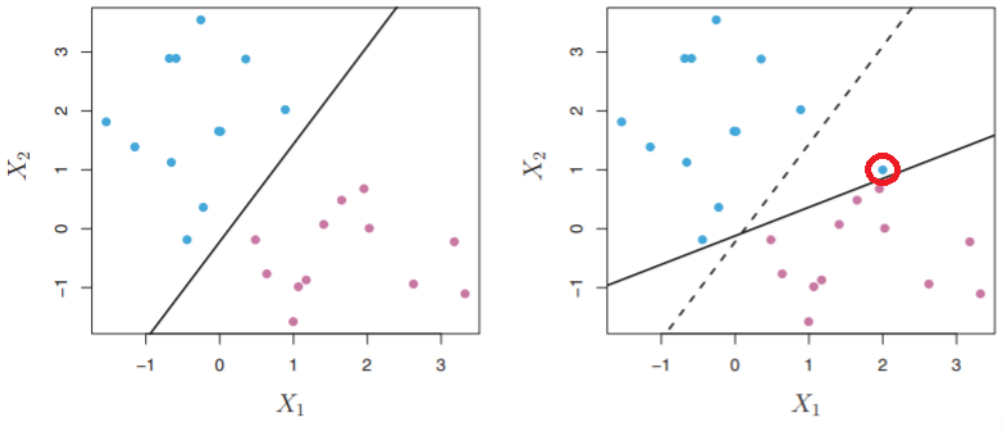
Esta propiedad ayuda a resolver el problema de optimización de manera más rápida, ya que en realidad solo algunas variables duales α_i serán no nulas. Normalmente se ocupan heurísticas para encontrar rápidamente que vectores no son de soporte y así resolver un problema mas pequeño (Buscar, por ejemplo, el método de Sequential Minimal Optimization)

6.4. Soft Margin

El planteamiento anterior tiene dos debilidades. Una, la más obvia, es que no siempre podemos tener datos separables y la segunda es que, incluso si los datos son linealmente separables, nuestro clasificador puede ser muy sensible a nuevos datos. Ver por ejemplo la siguiente imagen.

Como se ve en la imagen, la llegada del nuevo dato (encerrado en rojo) cambia drásticamente nuestro modelo.

Para solucionar estos problemas (parcialmente) agregamos variables de holgura, que nos permitirán clasificar mal algunos datos. La idea es que, sacrificando algunos datos (muy probablemente



outliers), podamos clasificar mejor la *mayoría* de los datos (Clasificarlos todos era claramente muy ambicioso) y así tener un modelo mas robusto.

Dicha formulazión del problema es la siguiente:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w||^2 + c\sum_{i=1}^{N}\xi_i \\ \text{s.a} \quad & y_i(w \cdot x_i + b) \geq 1 - \xi_i, \ i = 1, \ldots, N. \end{aligned}$$

donde c es un parámetro (Ya hablaremos de él).

Los ξ indican que tan mal clasificado está un punto. Si $\xi_i = 0$, entonces el dato x_i está al lado correcto del plano. Si $0 < \xi_i < 1$, entonces x_i está al lado correcto del plano, pero viola la restricción del margen. Finalmente si $\xi_i > 1$, entonces el punto esta al lado incorrecto del plano (Ver la siguiente figura)

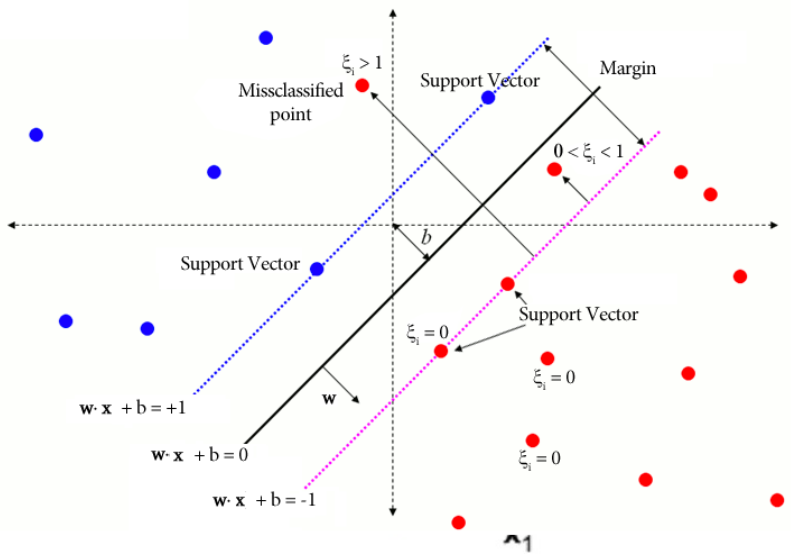


Figura 20: Visualización de los valores de ξ

El dual de este problema es (Usando Lagrangiano etc.)

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ \text{s.a} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq c \end{aligned}$$

Volveremos a este dual cuando veamos métodos no lineales.

Por último, ¿Qué hacer con el parámetro c? Guarda relación con el *bias-variance tradeoff*. La variable c es un parámetro de tolerancia. A mayor c, encontraremos un mayor márgen, lo que significa que nuestro modelo no estará sumamente ajustado a los datos (Lo que lo vuelve mas insesgado), pero probablemente se equivoque más (Tenga mayor varianza). Al revés, un c muy pequeño nos llevará más cerca del caso inicial, donde nuestro margen será más pequeño, y por ende, raramente violado (Poca varianza), pero nuestro modelo podría quedar sobreajustado a nuestros datos, logrando un mayor sesgo.En la práctica se usa *cross-validation* para encontrar un buen parámetro c.

6.5. Kernel Methods

Los métodos anteriores se ven interesantes, pero podría desmotivar el hecho que sólo funciona para datos linealmente separables (Con un poco de holgura en el caso de soft margin).

Un caso particular es el problema de clasificación XOR que se muestra en la siguiente figura

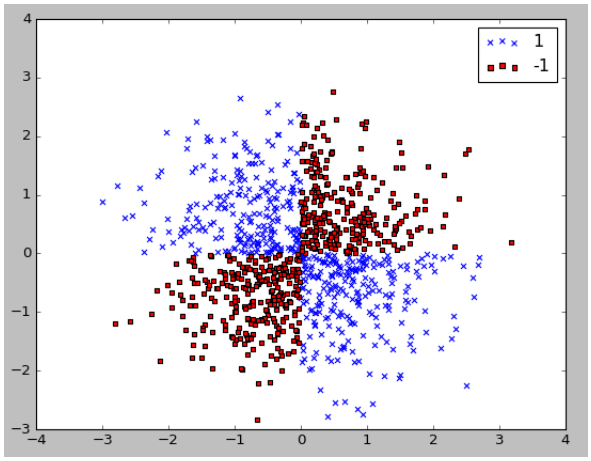


Figura 21: Datos XOR no son linealmente separables

Dichos datos no son linealmente separables en $\mathbb{R}^2$, pero si consideramos el siguiente mapeo a $\mathbb{R}^3$

$$\phi(x_1, x_2) = (x_1, x_2, x_1 x_2) \tag{218}$$

Entonces es claro que el plano $z = 0$ en $\mathbb{R}^3$ será capaz de separar dichos puntos (Pues si $x_1 x_2 > 0$ entonces dicho punto será rojo y estará por sobre el plano $z = 0$, y análogamente sucede algo similar con los puntos azules).

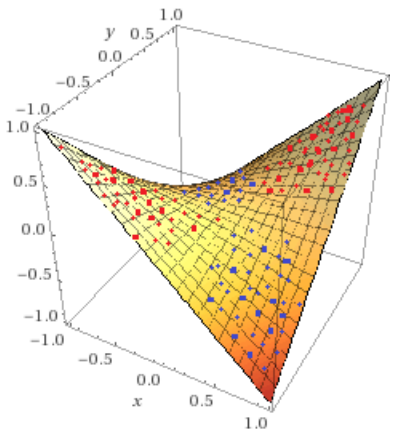


Figura 22: Los puntos rojos y azules corresponden a los datos mapeados a través de ϕ. El plano $z = 0$ es claramente capaz de separar puntos rojos y puntos auzles en este nuevo espacio.

La función ϕ se conoce normalmente como **feature map**. Es decir una función que entrega ciertos atributos de los datos (En este caso, por la forma del problema, un atributo importantes era justamente x_1x_2).

Dicha función puede ser muy díficil de definir en un problema particular, de hecho ya la función anterior dificilmente nos funcionará en otro problema que no sea XOR.

Sin embargo, muchas veces no nos interesa la forma explícita de la función, si no que más bien nos interesa trabajar con ella, en particular nos basta poder calcular $\phi(x)^T\phi(x)$(Ya vimos en las partes anteriores que dicho término, el producto punto de los datos, es relevante).

Para ello se ocupan los llamados **kernels**.

Def: Un (Mercer) Kernel es una función $K : X \times X \to \mathbb{R}$ tal que

- Es simétrica $K(x_1, x_2) = K(x_2, x_1)$
- Es definida positiva, es decir

$$\int_{X^2} K(x_1, x_2)g(x_1)g(x_2)dx_1dx_2 \geq 0$$

  para toda g continua.

Lema: Todo (mercer) Kernel se puede descomponer de la forma

$$K(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$$

donde $\phi : X \to \mathbb{R}^D$ donde D puede ser ∞.

Es decir un kernel nos permite evaluar el producto punto (Qué será todo lo que necesitemos) en un espacio de alta dimensión, sin necesidad de encontrar explícitamente el feature map. Esto es lo que se conoce en machine learning como el **kernel trick**.

Distintos tipos de kernels, implicarán distintos tipos de feature maps. Por ejemplo

- **Radial Basis Function kernel:** Este kernel se defne por

$$K_{rbf}(x_1, x_2) = \sigma^2 \exp\left(-\frac{(x_1 - x_2)^2}{2l^2}\right)$$

Las fronteras que entrega son suaves (infinitamente diferenciables) y tiene dos parámetros. El parámetro σ^2 es un parámetro de escala, y el parámetro l controla la oscilación de la curva.

- **Periodic kernel:**

$$K_{per}(x_1, x_2) = \sigma^2 \exp\left(-\frac{2\,\text{sen}^2(\pi|x_1 - x_2|/p)}{l^2}\right)$$

Este kernel es capaz de rescatar features periódicos en los datos (Controlados por el parámetro p). Los otros parámetros cumplen la misma función que el kernel anterior.

6.6. Ejemplo

Usemos el kernel trick para kernelizar un algoritmo ya conocido. EL rpoblema de ridge-regression consiste en que tienes un set de datos (y, X) con X una matriz e y un vector y queremos resolver

$$\min_{w\in\mathbb{R}^d} \frac{1}{n}||y - Xw||_2^2 + \lambda \sum_{i=1}^{d} w^2$$

donde λ es sólo un parámetro de regularización.
El resultado de este problema es explícito y dado por

$$\hat{w} = (X^T X + \lambda I)^{-1} X^T y$$

Para un nuevo punto (x', y') la predicción es:

$$\hat{y}' = y^T (XX^T + \lambda I)^{-1} X x'$$

Es aquí donde podemos ocupar el kernel trick, pues XX^T corresponde a los productos punto posibles entre las entradas de x. Podemos mapear a un espacio de alta dimensionaldiad cambiando a XX^T por una matriz K tal que

$$K_{ij} = K(x_i, x_j)$$

donde K es un kernel que nosotros estimamos conveniente. También el término Xx' lo cambiamos por un término k tal que $k_i = K(x_i, x')$.
Y así la predicción kernelizada queda

$$\hat{y}' = y^T (K + \lambda I)^{-1} k$$

Notar que una de las desventajas del método, a medida que entran más datos, deberemos invertir una matriz cada vez más grande.

6.7. Kernel SVM

Ahora los datos de entrenamiento los llevaremos a un espacio de dimensionalidad alta (donde si será posible separarlos linealmente) a través de un feature map $\phi(\cdot)$. Encontrar un feature map acorde es en general dificil e intractable.

El truco del kernel consiste en no tener que definir dicho feature map, si no que solo trabajar con el producto punto en dicho espacio de dimensionalidad alta el cual esta dado por un kernel

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

Dicho Kernel sí es conocido y tiene hiperparámetros los cuales podemos ajustar a nuestro problema. Para ello usaremos todo lo visto en las partes anteriores.

Mapeando los datos a través de ϕ y sin utilizar soft margin, el primal nos queda

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}||w||^2 + c\sum_{i=1}^{N} \xi_i \\ \text{s.a} \quad & y_i(w \cdot \phi(x_i) + b) \geq 1 - \xi_i,\ i = 1, \ldots, N. \end{aligned}$$

Pasandonos al dual, tendremos que

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N} \alpha_i \alpha_j y_i y_j < \phi(x_i), \phi(x_j) > \\ \text{s.a} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq c \end{aligned}$$

Notar que en este problema el único término donde aparece $\phi(x)$ es en el producto punto $\langle \phi(x_i), \phi(x_j) \rangle$ el cual es igual a $K(x_i, x_j)$. Esto es justamente el Kernel Trick.

El problema así nos queda

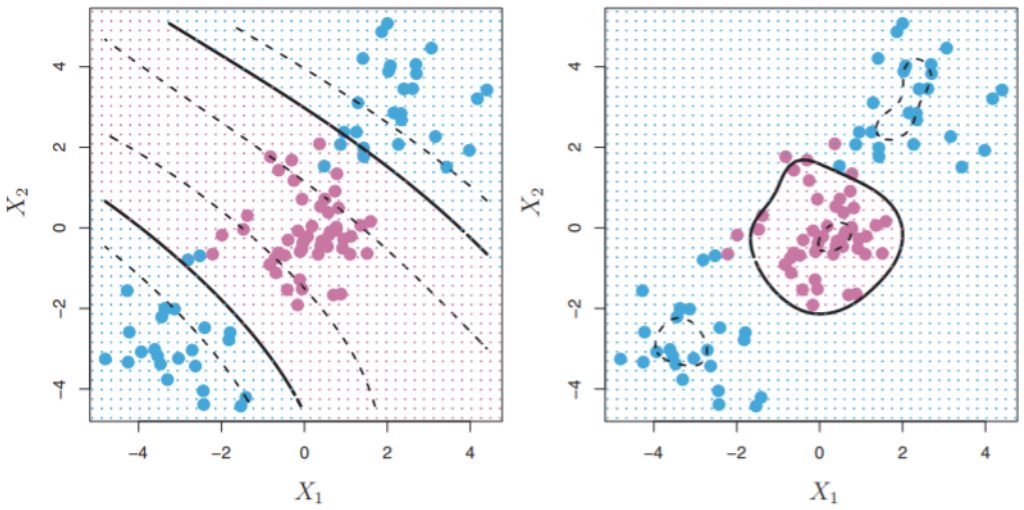
$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \\ \text{s.a} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq c \end{aligned}$$

El cual si es tratable: una vez calculados todos los $\binom{n}{2}$ productos de la forma $K(x_i, x_j)$, lo unico que queda es resolver un problema QP (El cual, por las condiciones de Mercer del kernel, tendrá solución, pues la matriz será definida positiva).

En la siguiente imagen vemos un ejemplo de SVM con dos kernels diferentes.

A la izquierda se ocupo un kernel polinomial de grado 3, dicho kernel permite un poco mas de curvatura en los límites del clasificador. A la derecha se ocupó un kernel RBF, el cual es capaz de encontrar fronteras de decisión más curvadas (Pero infinitamente diferenciables).

Tener en cuenta que esta manera de resolver el problema tiene una tremenda ventaja computacional. Mover los datos a un espacio más grande mediante explícitamente definiendo $\phi(x)$, puede



ser muy costoso, o incluso imposible en el caso que el espacio de llegada sea infinitamente dimensional. Usando funciones kernel dicha función queda implícita en el problema, dejandonos incluso trabajar en el caso infinito-dimensional (En el caso de RBF por ejemplo)

7. Procesos Gaussianos

Como se vio en capítulos anteriores, el problema de regresión busca encontrar una función $y = f(x)$, dado un conjunto de pares de la forma $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$. Dento de los métodos vistos para resolver el problema de regresión, se vio el de regresión lineal, lineal en los parámetros y no lineals. Una característica en común que tienen estos métodos es que el proceso de entrenamiento consiste en encontrar un número **fijo** de parámetros, que minimicen cierta función objetivo, donde la cantidad de parámetros y la forma del modelo es parte del diseño. A este tipo de modelos se les llama *paramétricos*.

En contraste, se encuentran los modelos *no paramétricos* los cuales **no** tienen un número fijo de parámetros, donde pueden llegar a ser en algunos casos infinito. Un ejemplo es el algoritmo k-vecinos más cercanos (KNN), donde los puntos del conjunto de entrenamiento son usados para clasificar nuevas muestras. Otro ejemplo son las máquinas de soporte vectorial (SVM) donde al entrenar se obtienen los vectores de soporte.

Es importante hacer la distinción entre parámetros que se aprenden y los parámetros de modelo, muchas veces llamados hyperparámetros, donde estos últimos pueden ser fijos independiente si el método es paramétrico y no paramétrico; esto se ve en el caso de SVM donde los parámetros serían los vectores de soporte, y los hyperparámetros serían el tipo de kernel, los parámetros de dicho kernel y los demás valores elegidos de antemano que definen el tipo de modelo.

En este capítulo introduciremos un método no paramétrico probabilístico de regresión no lineal, llamados Procesos Gaussianos ($\mathcal{GP}$), este en vez de encontrar un candidato único de la función a estimar, define una distribución sobre funciones $\mathbb{P}(f)$, donde $f(\cdot)$ es una función de un espacio de entrada $\mathcal{X}$ a los reales, $f : \mathcal{X} \to \mathbb{R}$. Esto tiene la virtud que permitirá cuantificar la incertidumbre puntual que existe en la predicción de nuestro modelo que servirá en forma de intervalos de confianza para la distribución Gaussiana.

Partiremos definiendo un $\mathcal{GP}$ como una distribución a priori sobre funciones, y mostraremos que la densidad posterior se puede encontrar de forma exacta y que esta también es un $\mathcal{GP}$, conservando sus propiedades.

$$\mathbb{P}(f|\mathcal{D}) = \frac{\mathbb{P}(f)\mathbb{P}(\mathcal{D}|f)}{\mathbb{P}(\mathcal{D})} \tag{219}$$

Es de notar que si $\mathcal{X}$ tiene cardinalidad infinita (por ejemplo $\mathcal{X} = \mathbb{R}$), $f(\cdot)$ puede ser visto como un vector infinito dimensional. Como en la práctica no podemos trabajar con un vector infinito dimensional, dados n puntos $\{x_i\}_1^n \subset \mathcal{X}$ podemos definirnos el vector n-dimensional de valores de la funciones evaluados en dichos puntos $f(\mathbf{x}) = [f(x_1), \ldots, f(x_n)]$.

Definición: Un Proceso Gaussiano ($\mathcal{GP}$) es una colección de variables aleatorias, tal que para cualquier subconjunto finito de puntos, estos tienen una distribución conjuntamente Gaussiana.

Al aplicar esta definición a nuestro caso anterior, $\mathbb{P}(f)$ será un $\mathcal{GP}$ y para cualquier conjunto finito $\{x_i\}_1^n \subset \mathcal{X}$, la distribución de $\mathbb{P}(f(\mathbf{x}))$ es Gaussiana multivariada $f(\mathbf{x}) = [f(x_1), \ldots, f(x_n)])$. En este caso las variables aleatorias representan el valor de la función $f(x_i)$ en la posición x_i.

Un $\mathcal{GP}$ queda completamente caracterizado por su función de media $m(\cdot)$ y función de covarianza $K(\cdot,\cdot)$, de esta forma para cualquier conjunto finito podemos encontrar la distribución. Definimos estas funciones como

$$m(x) = \mathbb{E}\left\{f(x)\right\} \tag{220}$$
$$K(x,x') = \mathbb{E}\left\{\left(f(x) - m(x)\right)\left(f(x') - m(x')\right)\right\} \tag{221}$$

Y de esta forma podemos escribir el proceso como:

$$f \sim \mathcal{GP}(m(\cdot), K(\cdot,\cdot)) \tag{222}$$

Donde para un conjunto finito tenemos que la marginal resulta de la forma:

$$f(\mathbf{x}) \sim \mathcal{N}(m(\mathbf{x}), K(\mathbf{x},\mathbf{x})) \tag{223}$$

Hasta el momento hemos hablado del espacio de entrada $\mathcal{X}$ como genérico, un caso común es definir los $\mathcal{GP}$ sobre el tiempo ($\mathbb{R}^+$), es decir que los x_i son instantes de tiempo. Es de notar que este no es el único caso, y se podría definir sobre un espacio más general, por ejemplo $\mathbb{R}^d$.

Otro punto a notar es que como estamos hablando de una colección (no necesariamente finita) de variables aleatorias, es necesario que se cumpla la propiedad de marginalización (o llamada consistencia[3]). Esta propiedad se refiere a que si un $\mathcal{GP}$ define una distribución multivariada para digamos dos variables $(y_1, y_2) \sim \mathcal{N}(\mu, \Sigma)$ entonces también debe definir $y_1 \sim \mathcal{N}(\mu_1, \Sigma_{11})$ donde μ_1 es la componente respectiva del vector μ y Σ_{11} la submatriz correspondiente de Σ. En otras palabras, el tomar un subconjunto más grande de puntos no cambia la distribución de un subconjunto más pequeño. Y podemos notar que esta condición se cumple si tomamos la función de covarianza definida anteriormente.

7.1. Muestreo de un prior $\mathcal{GP}$

Como fue mencionado, un $\mathcal{GP}$ define un *prior* sobre funciones, por lo que, antes de ver ningún dato se podría obtener una muestra de este proceso dado una función de media y covarianza. Un supuesto común es asumir la función de media $m(\cdot) = 0$ por lo que solo nos queda definir una función de covarianza o kernel, un ejemplo de kernel es el *Exponencial Cuadrático* (Square Exponential), también conocido como RBF (Radial Basis function) o Kernel Gaussiano (en general se evita este nombre porque este kernel no tiene relación con la distribución de los datos).

$$K_{SE}(x,x') = \sigma^2 \exp\left(-\frac{(x-x')^2}{2\ell^2}\right) \tag{224}$$

Donde en este caso los parámetros son interpretables (y como veremos más adelante pueden ser aprendidos a través de un conjunto de entrenamiento) donde σ^2 es la varianza de la función, notar que esta es la diagonal de la matriz covarianza. El parámetro ℓ es conocido como el *lenghtscale* que determina que tan lejos tiene influencia un punto sobre otro, donde en general un punto no

[3]Para más detalles puede ver el teorema de consistencia de Kolmogorov

tendrá influencia más allá de ℓ unidades alrededor.

Como sabemos, las funciones definidas por el $\mathcal{GP}$ son vectores infinito dimensionales, por lo que no podemos muestrear de toda la función, pero tomando una cantidad suficiente de puntos podemos graficar muestras de un $\mathcal{GP}$ dada una función de covarianza. Tomando un $\mathcal{GP}$ con media cero ($m(\cdot) = 0$) y kernel SE, muestras del proceso para distintos valores de ℓ obtenemos la Fig.23, donde el área sombreada corresponde al intervalo de confianza del 95 %. Se puede ver que el parámetro ℓ controla que tan erráticas son las funciones, donde a medida que va a aumentando las muestras se vuelven funciones más suaves.

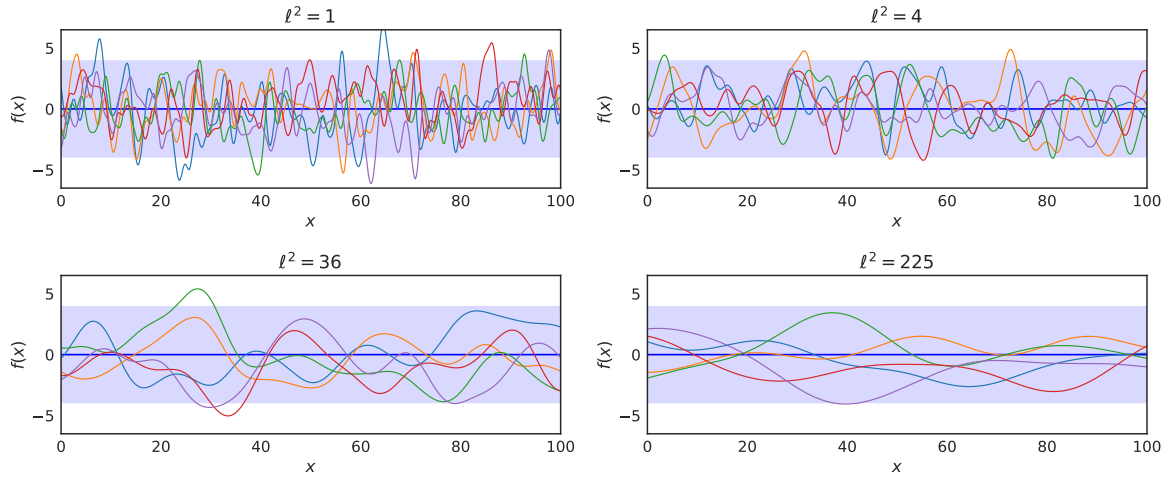


Figura 23: Muestras de un prior $\mathcal{GP}$ con kernel SE, para distintos *lenghtscales* (ℓ) y función media $m(\cdot) = 0$, la parte sombreada corresponde al intervalo de confianza del 95 %. Se puede ver que a mayor ℓ las funciones se van volviendo más suaves.

7.2. Incorporando Información: Evaluación sin ruido

Ahora que ya podemos muestrear de nuestro prior, nos interesaría incorporar las observaciones que tenemos de la función a nuestro modelo. Para esto, primero consideramos el caso en que las observaciones son sin ruido, es decir tenemos observaciones de la forma $\{x_i, f(x_i)\}_1^n$, donde tenemos el valor real de nuestra función en los puntos $[x_1, \ldots, x_n] = X$.

Luego, tenemos el par de entradas y observaciones $(X, f(X))$ digamos que queremos realizar una predicción en el conjunto X_* de n_* puntos, luego la distribución conjunta es de la forma:

$$\begin{bmatrix} f(X) \\ f(X_*) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix} \right) \tag{225}$$

Donde la submatriz $K(X, X_*)$ es de $n \times n_*$, $K(X, X)$ de $n \times n$ y así respectivamente para cada submatriz.

Pero a nosotros nos interesa encontrar la densidad posterior de $f(\cdot)$, y dada las observaciones y una función de covarianza podemos evaluar la verosimilitud, que también es Gaussiana, lo que nos lleva a un punto clave de los Processos Gaussianos.

Dado un prior $\mathcal{GP}$ sobre $f(\cdot)$ y una verosimilitud Gaussiana, la posterior sobre $f(\cdot)$ es también un $\mathcal{GP}$.

Luego, podemos condicionar sobre las observaciones $(X, f(X))$ y obtenemos

$$f(X_*)|f(X), X \sim \mathcal{N}(m_{X_*|X}, \Sigma_{X_*|X}) \tag{226}$$

Donde la media y covarianza son:

$$m_{X_*|X} = m(X_*) + K(X_*, X)K^{-1}(X, X)(f(X) - m(X)) \tag{227}$$

$$\Sigma_{X_*|X} = K(X_*, X_*) - K(X_*, X)K^{-1}(X, X)K(X, X_*) \tag{228}$$

Ahora, dado un conjunto de observaciones y dada una función de covarianza, podemos obtener la densidad posterior. Para mostrar esto tomemos el caso de hacer regresión para una función conocida, para la cual tenemos observaciones sin ruido muestreadas no uniformemente, con estas observaciones queremos encontrar la función real de las que provienen; para esto usamos un prior $\mathcal{GP}$ con función media nula y kernel SE (por el momento tendrá parámetros fijos), nos damos un rango donde queremos hacer predicción y condicionamos en las observaciones usando la Ec.(227). En este caso las observaciones corresponden al 15 % de los puntos generados por nuestra función sintética.

Esto se muestra en la Fig.24, donde podemos ver que la media de la posterior pasa por las observaciones sin incertidumbre asociada, es decir que para estos puntos se tiene una posterior degenerada pues no hay varianza. Se puede ver que a medida que la predicción se aleja de las observaciones el intervalo de confianza (al que lo podemos asociar con incertidumbre del modelo) va creciendo.

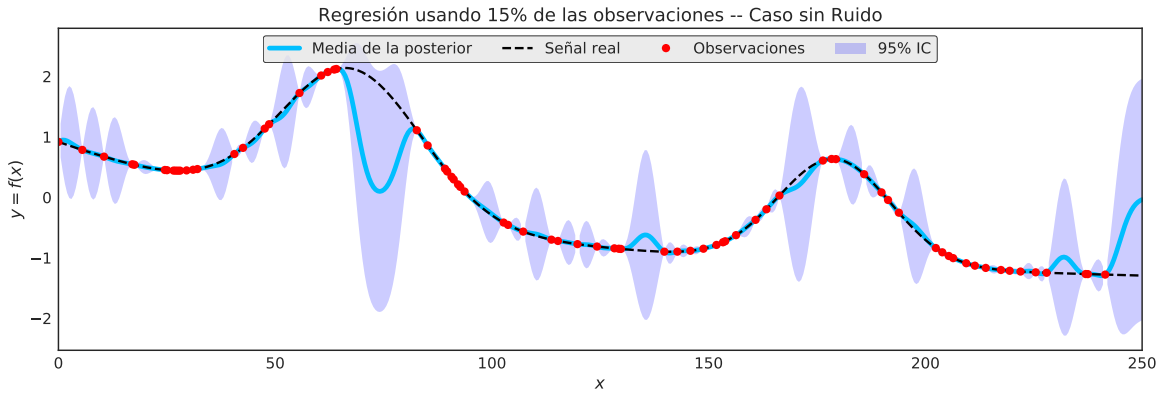


Figura 24: Regresión con $\mathcal{GP}$ para señal sintetica usando el 15 % de los datos muestreados de forma no uniforme, utilizand un $\mathcal{GP}$ de media nula y kernel SE.

7.3. Incorporando Información: Evaluación con ruido

En aplicaciones reales el caso de tener observaciones sin ruido es poco habitual, por lo que si queremos incorporar la incertidumbre real debemos tomar en cuenta errores de medición en nuestro modelo. Tomemos el caso de tener observaciones contaminadas con ruido i.i.d (independientes idénticamente distribuidas) donde las observaciones serán de la forma $y_i = f(x_i) + \eta$ donde $\eta \sim$

$\mathcal{N}(0, \sigma_n^2)$, donde ahora nuestro conjunto de observaciones es de la forma (X, Y) donde $Y = f(X) + \eta$.

Lo que en nuestro modelo equivale a agregar un término a la función de covarianza

$$cov(Y) = K(X, X) + \sigma_n^2 \mathbb{I} \tag{229}$$

Donde si tenemos el mismo caso anterior, observaciones (X, Y) y queremos evaluar en X_*, la conjunta queda

$$\begin{bmatrix} Y \\ f(X_*) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(X) \\ m(X_*) \end{bmatrix}, \begin{bmatrix} K(X, X) + \sigma_n^2 \mathbb{I} & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix} \right) \tag{230}$$

Notemos que el termino de ruido solo es agregado al subbloque correspondiente a las observaciones, no se agrega el termino en los otros subbloques pues buscamos hacer una predicción de la función latente $f(\cdot)$ y no una versión ruidosa de esta. Igual que en el caso sin ruido, podemos condicionar esta conjunta a las observaciones y obtenemos

$$f(X_*)|Y, X \sim \mathcal{N}(m_{X_*|X}, \Sigma_{X_*|X}) \tag{231}$$

Donde la media y covarianza son:

$$m_{X_*|X} = m(X_*) + K(X_*, X)[K(X, X) + \sigma_n^2 \mathbb{I}]^{-1}(Y - m(X)) \tag{232}$$

$$\Sigma_{X_*|X} = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 \mathbb{I}]^{-1} K(X, X_*) \tag{233}$$

Si tomamos el mismo ejemplo anterior, pero añadimos el ruido al modelo, obtenemos la predicción de la Fig.25. En este caso podemos ver que la media de la posterior no necesariamente coincide su valor con el de la observación, pues se toma en cuenta la incertidumbre en las observaciones mismas, también se ve que no se obtienen soluciones degeneradas incluso en zonas donde hay observaciones aglomeradas.

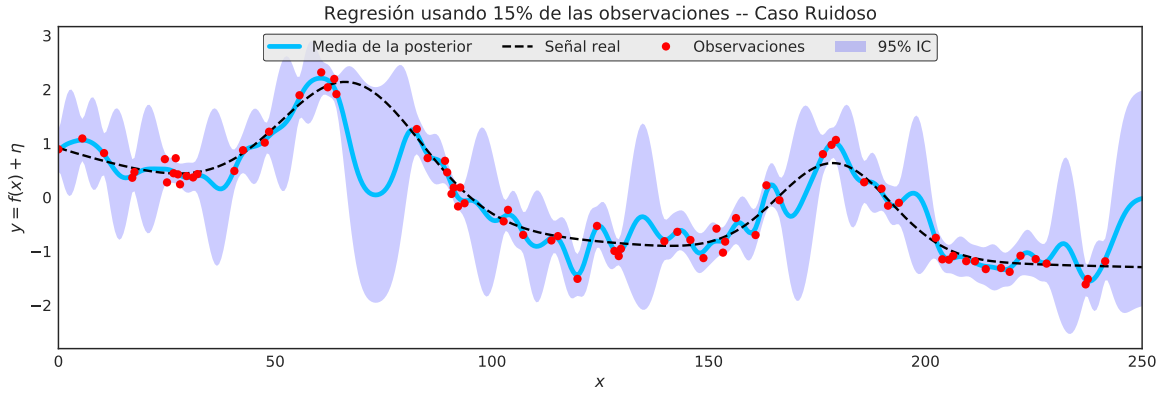


Figura 25: Regresión con $\mathcal{GP}$ para señal sintetica usando el 15 % de los datos muestreados de forma no uniforme y contaminados con ruido Gaussiano, utilizando un $\mathcal{GP}$ de media nula y kernel SE.

7.4. Entrenamiento y optimización de un $\mathcal{GP}$

hasta el momento nos hemos dado la función de covarianza y sus parámetros, por lo que aún no hemos hecho realizado el *aprendizaje* de nuestro $\mathcal{GP}$, que es lo que haremos a continuación.

¿Qué es entrenar un $\mathcal{GP}$?

Vimos que dada una función de covarianza podemos representar el proceso, y podemos encontrar analíticamente la densidad posterior de nuestra función $f(\cdot)$ condicionando a las observaciones. Pero la forma que tendrá la posterior y la función fuera de las observaciones dependerá fuertemente en nuestra función kernel escogida, en este sentido, para un kernel dado nos gustaría encontrar los parámetros de este que mejor representen nuestra función a estimar.

Nos referiremos a entrenar u optimizar un $\mathcal{GP}$ cuando queremos obtener los hyperparámetros, es decir los parámetros del kernel (los denotamos θ) y la varianza del ruido (la denotamos σ_n^2) si es que aplica.

Para esto nos gustaría poder comparar funciones de covarianza, o mejor aún un funcional que podamos optimizar, afortunadamente podemos usar la *verosimilitud marginal*, obtenida marginalizando sobre la función $f(\cdot)$, donde dado un conjunto de entrenamiento $(X, Y) = \{(x_i, y_i)\}_{i=1}^n$, esta dada por

$$\mathbb{P}(Y|X,\theta,\sigma) = \int \mathbb{P}(Y|f,X,\theta,\sigma_n)p(f|X,\theta,\sigma)df \tag{234}$$

$$= \frac{1}{(2\pi|\mathbf{K}_y|)^{\frac{n}{2}}} \exp\left(-\frac{1}{2}(Y-\mathbf{m})^T\mathbf{K}_y^{-1}(Y-\mathbf{m})\right) \tag{235}$$

Donde $\mathbf{m} = m(X)$ y $\mathbf{K}_y = K_\theta(X,X) + \sigma_n^2\mathbb{I}$, la matriz de covarianza dados los parámetros θ agregando el término de la diagonal correspondiente al ruido. De la misma forma que lo hacemos con otros modelos probabilísticos, en vez de maximizar la verosimilitud, en conveniente minimizar la log-verosimilitud negativa (NLL) dada por la expresión:

$$NLL = -\log \mathbb{P}(Y|X,\theta,\sigma_n) \tag{236}$$

$$NLL = \underbrace{\frac{1}{2}\log|\mathbf{K_y}|}_{\text{Penalización por complejidad}} + \underbrace{\frac{1}{2}(Y-\mathbf{m})^T\mathbf{K}_y^{-1}(Y-\mathbf{m})}_{\text{Data fit (Única parte que depende de } Y)} + \underbrace{\frac{n}{2}\log 2\pi}_{\text{Constante de normalización}} \tag{237}$$

De izquierda a derecha, el primer término tiene el rol de penalizar por la complejidad del modelo, y vemos que depende solo del kernel y las entradas; el segundo término cuantifica que tan bien se ajusta el modelo a los datos, y es la única componente que depende de las observaciones Y (ruidosas) de la función, el último término es una constante de normalización.

Siguiendo con los ejemplos anteriores, para el mismo conjunto de observaciones ruidosas, se calculan las tres componentes de la NLL, utilizando un kernel SE, para este caso, se dejan fijo tanto la varianza de la señal como la varianza del ruido y se varia el *lengthscale* ℓ del kernel, en la Fig.26.

Al ir variando ℓ se ve que la penalización por complejidad va disminuyendo, pues a mayor ℓ menos complejas son las funciones (recordar Fig.23, a mayor ℓ más

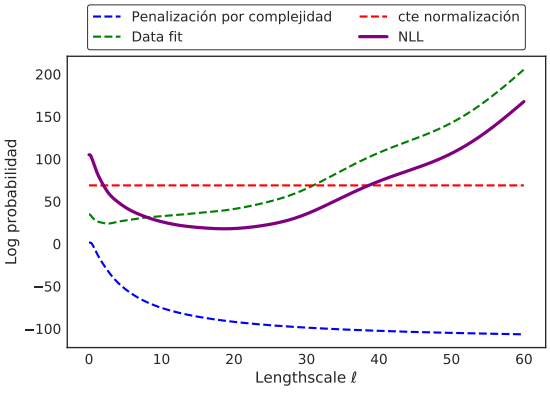


Figura 26: Log verosimilitud marginal ne-

suaves y regulares las muestras), también vemos que la componente del ajuste de datos comienza a decaer y luego se mantiene en incremento, pues a mayor *lengthscale* el modelo se vuelve menos flexible, por lo que no es capaz de ajustarse de manera correcta a los datos. De esta forma, el proceso de entrenamiento dará preferencia a funciones que se ajusten bien a los datos sin ser tan complejas, ahora viendo el valor total del NLL vemos que este alcanza su mínimo en ℓ en el rango de $[10 - 30]$.

Ya tenemos una noción de que es entrenar un $\mathcal{GP}$, queremos minimizar la NLL (Ec.(237)) y encontrar los parámetros del kernel y del ruido (si aplica). Ahora discutiremos formas de optimizar este funcional.

¿Cómo se entrena?

Como contamos con una expresión cerrada para la NLL, podemos utilizar métodos clásicos de optimización, una opción es calcular el gradiente de esta función objetivo y aplicar algún método basado en gradiente, como L-BFGS; otra es utilizar el método de Powell que no requiere que la función sea diferenciable, por lo que no utiliza gradiente.

Siguiendo ejemplos anteriores, usando la misma señal sintética y las mismas observaciones ruidosas de la Fig.25, ahora optimizamos nuestro $\mathcal{GP}$ utilizando L-BFGS, lo que nos entrega como resultado el mostrado en la Tabla.1 donde comparamos los parámetros del $\mathcal{GP}$ sin entrenar usado en la Fig.25, podemos ver que no difiere mucho en cuanto a las varianzas, pues como es una señal sintética poseíamos control sobre la varianza del ruido, el valor obtenido luego de optimizar es cercano al valor real (0.2), vemos que el *lengthscale* óptimo es concordante con lo analizado en la Fig.26.

Por último, podemos ver la predicción usando este $\mathcal{GP}$ optimizado en la Fig.27 donde vemos que se tiene una del proceso más concordante con la función real, esto se ve especialmente en la zona con pocas observaciones, entre 50 y 100 para x, donde el $\mathcal{GP}$ sin entrenar presentaba mucha incertidumbre en esa zona, en cuanto ahora se tiene un intervalo de confianza más bien limitado pues la señal no era muy compleja.

	σ_{ruido}	ℓ	$\sigma_{\text{señal}}$	NLL
Sin entrenar	0.2	3.1622	1	**55.3538**
Entrenado	0.2067	18.7267	0.9956	**17.6945**

Cuadro 1: Resultado optimización $\mathcal{GP}$ y comparación con los parámetros sin entrenar.

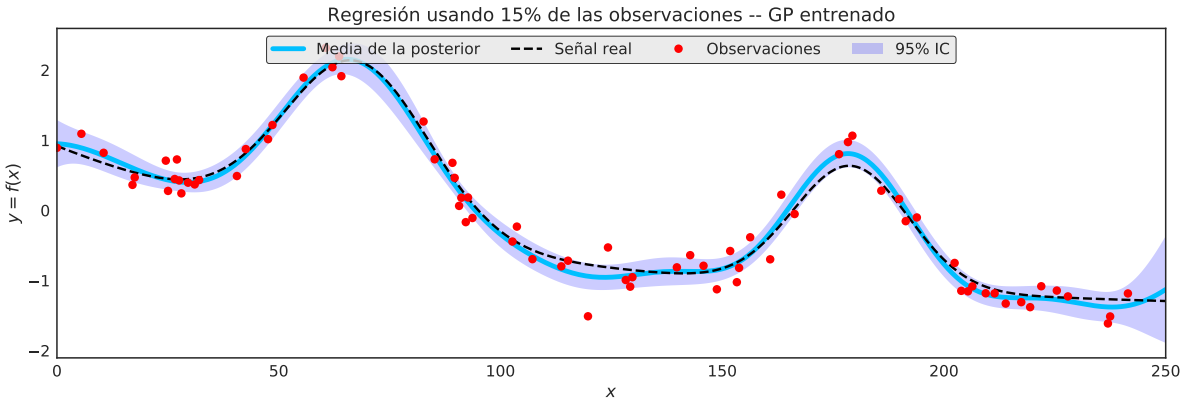


Figura 27: Regresión con $\mathcal{GP}$ para señal sintetica usando el 15 % de los datos muestreados de forma no uniforme y contaminados con ruido Gaussiano, utilizando un $\mathcal{GP}$ de media nula y kernel SE; Modelo optimizado utilizando L-BFGS.

La desventaja de utilizar métodos tradicionales de optimización es que estos entregan una solución puntual (en el sentido de un valor para cada parámetro, el $\mathcal{GP}$ sigue siendo una distribución sobre funciones), en distintas aplicaciones puede que un valor fijo de parámetros no represente de manera idónea el proceso latente a estudiar. Una opción sería computar la densidad posterior de los parámetros del kernel condicionado a las observaciones. Lamentablemente en la mayoría de los casos no existe una expresión cerrada para esta posterior, por lo que debemos recurrir a métodos de *inferencia aproximada* como Markov Chain Monte Carlo (MCMC) o Inferencia Variacional.

7.4.1. Complejidad Computacional

Es importante reconocer una de las principales desventajas de utilizar un $\mathcal{GP}$ cuando se cuenta con una gran cantidad de datos, esto es, su costo computacional. Recordando, cuando queremos entrenar nuestro $\mathcal{GP}$ vamos a minimizar la log verosimilitud marginal negativa (NLL), mostrada en la Ec.(237), donde al observar en segundo término vemos que es la operación más costosa siendo $\mathcal{O}(n^3)$ con respecto al número de puntos de entrenamiento n. Hay que tomar en cuenta que la evaluación de la NLL se debe hacer en cada iteración del método de optimización elegido.

En cuanto a la evaluación, esta está dada por la Ec.(232), donde en este caso vemos que es de orden cuadrático $\mathcal{O}(n^2)$ con respecto al número de puntos de test. Es de notar que esta vez no tomamos en cuenta la inversa de la matriz de Gram del conjunto de entrenamiento, pues puede ser precalculada y ser usada para múltiples conjuntos de test.

Lo anterior mencionado limita la cantidad de problemas que se pueden abordar usando estos procesos, sin embargo existen formas de obtener aproximaciones que siguen manteniendo la estructura y deseables propiedades de los $\mathcal{GP}$ a un menor costo computacional, estos reducen el orden de $\mathcal{O}(n^3)$ a $\mathcal{O}(nm^2)$ en entrenamiento y $\mathcal{O}(m^2)$ en test, con $m < n$. Un ejemplo de estos son los *Sparse Gaussian Processes* que utilizan una aproximación de rango bajo para la matriz de covarianza, utilizando *pseudo-inputs* en vez el conjunto de entrenamiento completo.

7.5. Funciones de covarianza (Kernels)

Una función de covarianza es una función de dos argumentos donde cualquier matriz que se obtiene como resultado en la evaluación de un conjunto de puntos (la llamaremos matriz de Gram)

es semidefinida positiva.

Hasta el momento solo hemos utilizado un tipo de función de covarianza, el llamado kernel *Squared Exponential* (SE), conocido también como kernel RBF, dado por la Ec.(238). En esta sección mostraremos distintos tipos de funciones de covarianza y los distintos tipos de funciones que generan.

$$K_{SE}(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right) \tag{238}$$

Es importante denotar tipos de familias de funciones de covarianza, si la función solo depende de la diferencia, es decir $k(x, x') = k(x - x')$ se le llamará *kernel estacionario*, más aún, si depende solo de la norma de la diferencia $k(x, x') = k(|x - x'|)$ se le llamará *kernel isométrico*, un ejemplo de esto es el kernel SE.

Es de notar que kernels estacionarios hacen que la covarianza entre puntos sea invariante a traslaciones en el espacio de entradas. Una noción importante es que un kernel puede ser visto como una medida de similaridad entre puntos, y en el caso de kernels estacionarios, mientras más cercanos estén dos puntos más símiles serán.

Kernel *Rational Quadratic* (RQ)

Este kernel viene dado por la Ec.(239), este puede ser interpretado como una suma infinita de kernels SE con distintos *lenghtscale*, donde α es un parámetro de variación de escala, notar que cuando $\alpha \to \infty$ el kernel tiende a uno SE. En la Fig.28 se muestra el kernel RQ, a la izquierda se ve el valor de la covarianza en función de la diferencia de los argumentos $x - x'$, a la derecha se muestran diferentes muestras de funciones con este kernel.

$$K_{RQ}(x, x') = \sigma^2 \left(1 + \frac{(x - x')^2}{2\alpha\ell^2}\right)^{-\alpha} \tag{239}$$

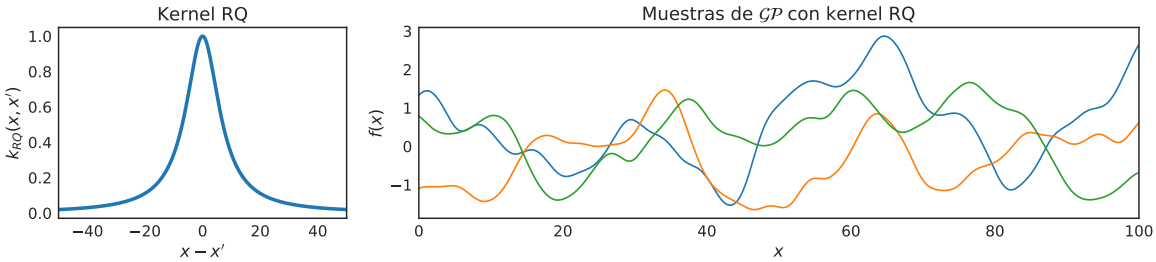


Figura 28: Kernel *Rational Quadratic*, en la izquierda se muestra la covarianza en función de su argumento $\tau = x - x'$, a la derecha de un $\mathcal{GP}$ usando un kernel RQ.

Kernel Periódico

Como su nombre lo indica, este kernel, dado por la Ec.(240), permite modelar funciones periódicas, donde el parámetro p controla el periodo de la función. Una extensión de este el kernel

localmente periódico, dado por la Ec.(241) que es un kernel SE ponderado por uno periódico, este da la libertad de tener funciones con una estructura periódica local.

$$K_P(x, x') = \sigma^2 \exp\left(-\frac{2\sin^2(\pi|x - x'|/p)}{\ell^2}\right) \tag{240}$$

$$K_{LP}(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right) \exp\left(-\frac{2\sin^2(\pi|x - x'|/p)}{\ell^2}\right) \tag{241}$$

En la Fig.29 se muestra el kernel periódico, a la izquierda se ve el valor de la covarianza en función de la diferencia de los argumentos $x - x'$, a la derecha se muestran diferentes muestras de funciones con este kernel.

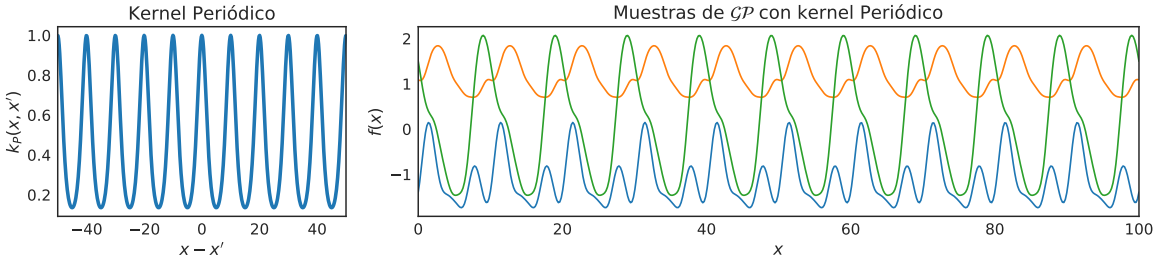


Figura 29: Kernel periódico, en la izquierda se muestra la covarianza en función de su argumento $\tau = x - x'$, a la derecha de un $\mathcal{GP}$ usando un kernel periódico.

7.5.1. Operaciones con kernels

A la hora de diseñar un $\mathcal{GP}$ y elegir una función de covarianza, no se está completamente limitado a kernels conocidos, sino que también se pueden combinar para obtener distintas funciones de covarianza y representar mejor el proceso.

Tanto la suma y multiplicación de kernels da como resultado un kernel válido que puede ser utilizado, también la exponencial de un kernel es también es un kernel, es decir $\exp(k_1(\cdot, \cdot))$ con k_1 un kernel válido.

7.5.2. Representación espectral

Un teorema importante para las funciones de covarianza en procesos débilmente estaicionarios es el teorema de Wiener–Khinchin, el cual dice que si para un proceso débilmente estacionario existe una función de covarianza $k(\tau)$ finita y definida para cualquier $\tau = x - x'$, entonces existe una función $S(\xi)$ tal que:

$$k(\tau) = \int S(\xi) e^{2\pi i \xi \cdot \tau} d\xi, \quad S(\xi) = \int k(\tau) e^{-2\pi i \xi \cdot \tau} d\tau \tag{242}$$

Donde i es la unidad imaginaria. $S(\xi)$ es conocida como la densidad espectral de potencia (PSD), en otras palabras, la función de covarianza $k(\tau)$ y la PSD $S(\xi)$ son duales de Fourier el uno del otro. Esto es extremadamente útil a la hora de diseñar funciones de covarianza pues este puede ser llevado a cavo en el dominio de la frecuencia y luego llevado a una función de covarianza usando la transformada inversa de Fourier.

7.6. Extensiones para un $\mathcal{GP}$

A continuación veremos un número de extensiones que se le pueden hacer a un $\mathcal{GP}$ para abordar distintos tipos de problemas.

7.6.1. $\mathcal{GP}$ de clasificación

Hasta el momento hemos visto como usar un $\mathcal{GP}$ para regresión, pero este también puede usado para clasificación, para esto simplemente "pasamos" nuestro $\mathcal{GP}$ por una función logística, para así obtener un prior sobre $\sigma(f(x))$ donde σ es la función logística. Sin embargo esto trae consigo un problema, pues ahora la distribución posterior a las observaciones no se tiene de forma analítica como para el caso de regresión, esto lleva a que tengamos que recurrir a métodos aproximado de inferencia. Una solución simple es utilizar la aproximación de Laplace, pero si se quieren aproximaciones más fidedignas métodos más complejos pueden ser usados como *Expectation Maximization* y métodos MCMC. En la Fig.30 se muestra un ejemplo con datos sintéticos, a diferencia de la mayoría de los clasificadores vistos, este entrega naturalmente una densidad de probabilidad para la función de decisión.

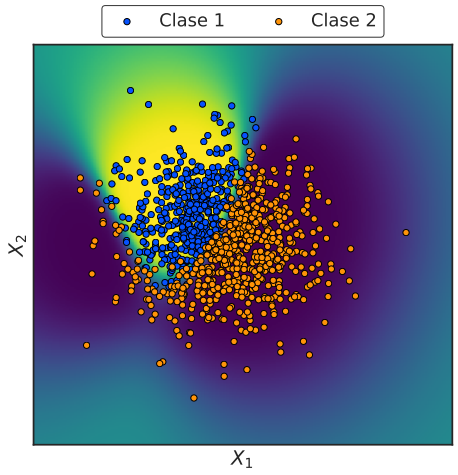


Figura 30: $\mathcal{GP}$ de clasificación utilizando datos sintéticos. Este clasificador entrega una densidad de probabilidad en vez de una sola función de decisión.

7.6.2. Selección automática de relevancia (ARD) (*Selección automática de features*)

Un $\mathcal{GP}$ define una densidad de probabilidad sobre funciones, donde estas funciones son del tipo $f : \mathbb{R}^D \to \mathbb{R}$, con D es finito, este es nuestra dimensión de entrada o "características". Haciendo un pequeño cambio en nuestra función kernel podemos hacer que esta automáticamente seleccione las entradas más relevantes con el problema, es decir realice una selección de características automática.

Si tomamos el kernel de la Ec.(243) vemos que es una multiplicación de kernels SE, donde se tiene un *lenghtscale* por cada entrada ℓ_d, sabemos que mientras más grande es este ℓ_d menos flexible será el $\mathcal{GP}$ respecto a cambios en ese eje, haciendo que las funciones del proceso dependan cada vez menos de la componente d a medida que $\ell_d \to \infty$. De esta forma se puede controlar de forma automática la relevancia de cada eje del conjunto de entrada, pues los parámetros del kernel se obtienen en el entrenamiento. De esta forma estamos optimizando también en que grado afecta cada variable en nuestra predicción.

$$k(x, x') = \sigma^2 \exp\left(-\sum_{d=1}^{D} \frac{(x_d - x'_d)^2}{2\ell_d^2}\right) \tag{243}$$

7.6.3. Multi output $\mathcal{GP}$

Hasta el momento solo hemos hablado de $\mathcal{GP}$ cuando nuestro proceso es solo una dimensión de salida. Se pueden extender los procesos Gaussianos a funciones de más de una salida o canal, donde ahora la función de covarianza $k(x, x')$ no entrega un escalar sino una matriz definida positiva, donde la diagonal corresponde a la covarianza del canal o autocovarianza y los elementos fuera de la diagonal corresponden a las covarianzas cruzadas o cross-covarianza. Este tipo de procesos Gaussianos aumentan considerablemente de complejidad al diseñar funciones de covarianza.

Dado un número m de canales, se tendrán m funciones de autocovarianza y ahora $m(m-1)/2$ funciones de covarianza y $k(x, x')$ será una matriz de $m \times m$. El desafio está en diseñar o escoger estas funciones de tal forma que para cualquier par de puntos x, x' la matriz $k(x, x')$ sea definida positiva.

Una opción simple es asumir que los canales son independientes entre sí, lo que equivale a entrenar independientemente m procesos Gaussianos, uno para cada canal, esto facilita el diseño de las funciones de covarianza pero hace que se pierdan relaciones entre los canales.

7.7. Diferentes Interpretaciones de un $\mathcal{GP}$

7.7.1. De regresión lineal a $\mathcal{GP}$

Partiendo de la regresión lineal donde el modelo es:

$$y_i = wX_i + \epsilon_i \tag{244}$$

Donde $\epsilon_i \sim \mathcal{N}(0, \sigma_n^2)$, luego podíamos extender este modelo usando M funciones base ϕ_m y obtener:

$$y_i = \sum_{m=1}^{M} w_m \phi(x_i) + \epsilon_i \tag{245}$$

El paso siguiente es la regresión Bayesiana en base de funciones, donde agregamos un prior sobre los pesos $w_m \sim \mathcal{N}(0, \lambda_m^2)$, donde si obtenemos la covarianza de este proceso y marginalizamos por los pesos w_m obtenemos:

$$Cov(x, x') = k(x, x') = \sum_{m=1}^{M} \lambda_m^2 \phi_m(x)^T \phi_m(x') + \delta_{x,x'} \sigma_n^2 \tag{246}$$

Donde $\delta_{x,x'}$ el delta de Kronecker. Si nos damos cuenta esto define un $\mathcal{GP}$ con la función covarianza con un número finito de funciones bases. En general los $\mathcal{GP}$ corresponderán a covarianzas con infinitas funciones bases, como veremos a continuación.

Tomando un modelo sin ruido, con un número M de funciones base ϕ_m, donde sobre los pesos se define un prior i.i.d $w_m \sim \mathcal{N}(0, \sigma^2)$, tenemos:

$$f(x) = \sum_{m=1}^{M} w_m \phi(x) \tag{247}$$

Y tomando $\phi = \exp(-\frac{1}{2\ell^2}(x-c_i)^2)$ donde c_i son los centros de estas bases, y luego haciendo tender el número de funciones base M a infinito tenemos que la covarianza es:

$$\mathbb{E}\{f(x)f(x')\} = \sigma^2 \sum_{m=1}^{M} \phi_m(x)\phi_m(x') \qquad \text{tomando } M \to \infty \tag{248}$$

$$\mathbb{E}\{f(x)f(x')\} \to \sigma^2 \int e^{-\frac{1}{2\ell^2}(x-c)^2} e^{-\frac{1}{2\ell^2}(x'-c)^2} dc \tag{249}$$

$$= \sigma^2 \sqrt{\pi\ell^2} e^{-\frac{1}{4\ell^2}(x-x')^2} \tag{250}$$

$$= k_{SE}(x, x') \tag{251}$$

Donde vemos que efectivamente el kernel SE es una función de covarianza para una composición infinita de funciones base.

7.7.2. Nota sobre RKHS

Dado un conjunto de entrenamiento $(X, Y) = \{x_i, y_i\}_{i=1}^n$ condicionando el proceso solo a una muestra de test $X_* = x_*$ y asumiendo una función media nula, de la Ec.(227) para un $\mathcal{GP}$ la posterior de la media está dada por:

$$\bar{f}(x_*) = m_{x_*|X} = k(X, x_*)(k(X, X) + \sigma_n \mathbb{I})^{-1} Y \tag{252}$$

Y tomamos el vector $\boldsymbol{\alpha} = (k(X, X) + \sigma_n \mathbb{I})^{-1} Y$ obtenemos la expresión:

$$\bar{f}(x_*) = \sum_{i=1}^{n} \alpha_i k(x_i, x_*) \tag{253}$$

Donde, a pesar de que el proceso esta descrito por (posiblemente infinita) funciones base, aún así es la suma de finitos términos, cada uno centrado en un punto de entrenamiento, esto es debido al teorema del representante de los Espacios de Hilbert de Kernel Reproductor (RKHS). La intuición detrás de esto es que incluso si el $\mathcal{GP}$ induce una distribución conjunta sobre todos los $y = f(x)$, una para cada x en el dominio, al hacer predicciones en el punto x_* solo nos interesa la distribución $(n+1)$-dimensional definida por los puntos de entrenamiento más este punto de test.

Resumen

¿Que es un Proceso Gaussiano?

Un Proceso Gaussiano ($\mathcal{GP}$) es un método no paramétrico que define un *prior sobre funciones*, lo cual se traduce en una *posterior* sobre funciones si la verosimilitud es Gaussiana. Este proceso se encuentra caracterizado completamente por su función de media $m(\cdot)$ (generalmente se asume 0) y función de covarianza o kernel $K_\theta(\cdot,\cdot)$ que puede depender o no de parámetros (θ).

¿Cómo se entrena?

Dado un conjunto de entrenamiento $(X, Y) = \{(x_i, y_i)\}_{i=1}^n$, donde y_i pueden estar contaminadas o no por ruido (generalmente se asume ruido Gaussiano de varianza σ_n^2) se *minimiza* la log-verosimilitud marginal negativa (NLL), que para el caso de $m(\cdot) = 0$ es:

$$NLL = -\log \mathbb{P}(Y|X, \theta, \sigma_n) \tag{254}$$

$$NLL = \underbrace{\frac{1}{2}\log|\mathbf{K_y}|}_{\substack{\text{Penalización}\\ \text{por}\\ \text{complejidad}}} + \underbrace{\frac{1}{2}Y^T\mathbf{K}_y^{-1}Y}_{\text{Data fit}} + \underbrace{\frac{n}{2}\log 2\pi}_{\substack{\text{Constante de}\\ \text{normalización}}} \tag{255}$$

Donde $\mathbf{K}_y = K_\theta(X, X) + \sigma_n^2\mathbb{I}$.

¿Cómo se evalúa en nuevos puntos?

Si queremos evaluar en el conjunto de test X_*, la evaluación es:

$$f(X_*)|Y, X \sim \mathcal{N}(0, \Sigma_{X_*|X}) \tag{256}$$

Donde la media y covarianza son:

$$m_{X_*|X} = K(X_*, X)[K(X, X) + \sigma_n^2\mathbb{I}]^{-1}Y \tag{257}$$

$$\Sigma_{X_*|X} = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2\mathbb{I}]^{-1}K(X, X_*) \tag{258}$$

¿Qué costo tiene?

- Entrenamiento: $\mathcal{O}(n^3)$ respecto al número de muestras del conjunto de **entrenamiento**.
- Evaluacion: $\mathcal{O}(n^2)$ respecto al número de muestras del conjunto de **test**.

Notas sobre la implementación:

Como la matriz de Gram ($K(X, X)$) es definida positiva y simétrica, para calcular su inversa de forma más eficiente se utiliza la descomposición de Cholesky, donde si $K = LL^T$, entonces $K^{-1} = (L^T)^{-1}L^{-1}$

8. Aprendizaje No Supervisado

8.1. Reducción de dimensionalidad

8.1.1. Principal Component Analysis (PCA)

Formalmente, consideremos una matriz de datos $\hat{X}$:

$$\hat{X} = \begin{bmatrix} \hat{x}_{11} & \dots & \hat{x}_{1m} \\ \vdots & \ddots & \vdots \\ \hat{x}_{n1} & \dots & \hat{x}_{nm} \end{bmatrix},$$

donde cada columna representa una variable de proceso distinta (característica) y cada fila un instante de tiempo, observación o ejemplo. De este modo se tienen m características, y n ejemplos. Es altamente recomendable que los datos de PCA estén normalizados, de modo que la normalización estará dada por:

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

La dirección de mayor variabilidad está dada por los vectores propios de la *matriz de covarianza empírica*:

$$\Sigma = \frac{1}{n-1}\hat{X}^T\hat{X} = V\Lambda V^T \tag{259}$$

donde V es la matriz donde cada columna es un vector propio y Λ es la matriz de valores propios. Un mayor valor para un valor propio indica mayor variabilidad para el vector propio asociado. Finalmente, los datos proyectados se pueden escribir como:

$$X_{\text{proy}} = \hat{X}V \tag{260}$$

Notemos que la transformación PCA es una rotación lineal de los datos.

Como observación, PCA encuentra una nueva base ortogonal tal que las componentes maximicen la variabilidad, esto implica que se pierde la intepretabilidad de las nuevas características generadas. A pesar de esto, utilizando las primeras 2 o 3 componentes PCA se pueden visualizar datos de alta dimensionabilidad.

Como se dijo anteriormente, es recomendable que las variables estén normalizadas, esto tiene justificación principalmente en evitar que las variables se vean afectadas por efectos de la escala. Por ejemplo, si tenemos dos variables temperatura en un rango de 0°C a 100°C, mientras que otra variable es la altura que va de 0m a 10m, notamos que claramente la variable temperatura aportará mayor variabilidad al sistema.

8.1.2. Kernel PCA

El método Kernel PCA es similar a PCA, pero esta vez se utiliza el truco del kernel para proyectar los datos. En ese sentido, en vez de calcular la matriz de covarianza empírica, se utiliza la matriz de Gram.

$$K_{ij} = K(x_i, x_j) = \phi(x_i)\phi(x_j)^T$$

Luego, se prosigue de la misma forma que PCA linear.

En la figura 31 se puede observar un ejemplo en que el resultado de PCA linear no es suficiente, puesto que el problema es simétrico, mientras que KPCA realiza una correcta separación de ambos clusters.

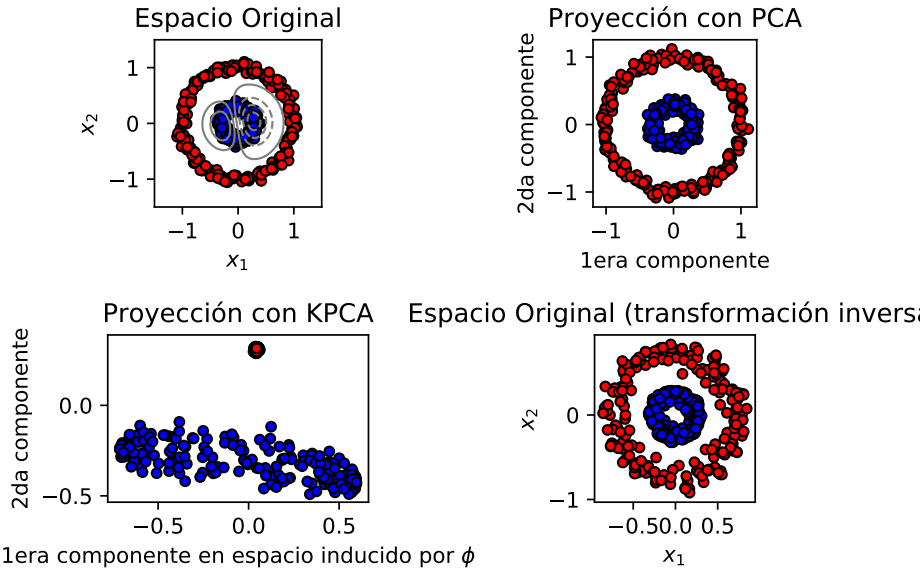


Figura 31: Ejemplo en que kernel PCA sobre un conjunto de datos que no es linealmente separable.

8.1.3. Probabilistic PCA

PCA probabilístico (PPCA, por su sigla en inglés) tiene su inspiración en que PCA se puede expresar como la solución via máxima verosimilitud de un modelo probabilístico de variable latente. De este modo, PPCA propone un método iterativo para obtener la solución evaluando solo cierto número de componentes, sin necesidad de calcular la matriz de covarianza empírica.

El modelo probabilístico para PCA en que se inspira PPCA es el siguiente:

Sea $x_i \in \mathbb{R}^m$ los elementos observados, inputs o variables y $z \in \mathbb{R}^l$ una variable latente explícita correspondiente al espacio de las componentes principales. Se define un prior para z:

$$p(z) = N(0, I) \tag{261}$$

De este modo, la distribución condicional de x dado z también es Gaussiana:

$$p(x|z) = N(Wz + \mu, \sigma^2 I) \tag{262}$$

Donde $W \in \mathbb{R}^{M \times l}$ y $\mu \in \mathbb{R}^m$ son parámetros a determinar. Notemos que no se pierde generalidad tomar el prior para z con media cero y varianza unitaria, puesto que si se toma otro prior más general, se produce el mismo modelo.

Dado que tenemos modelo paramétrico probabilístico, podemos estimar los parámetros con máxima verosimilitud. Dado los datos $X = \{x_i\}_{i=1}^n$. La log-verosimilitud está dada por:

$$\log p(X|W,\mu,\sigma^2) = \sum_{i=1}^{n} \log p(x_i|W,\mu,\sigma^2) \tag{263}$$

$$= \frac{nl}{2}\log(2\pi) - \frac{n}{2}log|C| - \frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)^T C^{-1}(x_i-\mu), \tag{264}$$

con $C = WW^T + \sigma^2 I$.
Usando la condición de primer orden obtenemos

$$\mu = \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{265}$$

De esta manera tenemos la función de log-verosimilitud completa:

$$\log p(X,Z|W,\mu,\sigma^2) = \sum_{i=1}^{n}\{\log p(x_i|z_i) + \log p(z_i)\} \tag{266}$$

y evaluando en $\mu = \bar{x}$

$$\begin{aligned}\mathbb{E}[p(X,Z|W,\mu,\sigma^2)] = -\sum_{i=1}^{n}\Bigg\{ & \frac{l}{2}\log(2\pi\sigma^2) \\ & + \frac{1}{2}\mathrm{Tr}(\mathbb{E}[z_i z_i^T]) \\ & \frac{1}{2\sigma^2}||x_i-\mu||^2 - \frac{1}{\sigma^2}\mathbb{E}[z_i]^T W^T(x_i-\mu) \\ & \frac{1}{2\sigma^2}\mathrm{Tr}(\mathbb{E}[z_i z_i^T]W^T W)\Bigg\}\end{aligned} \tag{267}$$

9. Clustering

9.1. k-means

Dado un entero $k \in \mathbb{N}$ y uin conjunto de observaciones $X = \{x_i\}_{i=1}$ con $x_i \in \mathbb{R}^D$ queremos separar los datos en k graupos, donde cada grupo se le asigna un centroide μ_k y cada elemento x_i se le asigna el grupo que tenga el centroide más cercano.

Sea r_{ik} la asignación, esta estará definida por:

$$r_{ik} = \begin{cases} 1 & \text{si } k = \text{argmin}||x_i - x_k|| \\ 0 & \text{si no.}\end{cases}$$

Es decir, para encontrar los centroides se debe minimizar la función:

$$J = \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} ||x_i - x_j||^2 \tag{268}$$

Para minimizar esta función utilizaremos un enfoque llamado *Expectation-Maximization*. Este es un método iterativo y como tal, tiene problemas con mínimo locales, pero para solucionar esto, basta inicializar el algoritmo muchas veces.

El algoritmo está dado por:

- **E-step:** En este paso, se calculan (actualizan) las asignaciones r_{ik}, dejando fijos μ_k. Lo que corresponde a asignar el dato x_i al centroide más cercano.
- **M-step:** El siguiente paso corresponde a actualizar los centroides μ_k dejando fijo las asignaciones r_{ik}.

  Como J es cuadrática en μ_k, entonces podemos utilizar la condición de primer orden:

  $$\mu_k = \frac{\sum_{i=1}^{N} r_{ik} x_i}{\sum_{i=1}^{N} r_{ik}} \tag{269}$$

  Lo que corresponde a asignar el centro del cluster al promedio de todas las muestras asignadas al antiguo cluster.

Ejemplo: En la figura 32 se observa un ejemplo de clustering utilizando kmeans. Como se puede notar, los clusters creados por kmeans son circulares, puesto que se utiliza distancia euclediana hace el centro del cluster.

9.2. Gaussian Mixture Model

La mezcla de gaussianas (GMM por su sigla en inglés) es un caso general de k-means, en donde no solo ajustamos los centros (vector de medias μ_k) si no también el modelo considera matrices de covarianza Σ_k. En ese sentido, se puede interpretar k-means como el caso particular en que $\Sigma_k = I$ y asignación directa (*hard labeling)*.

En el modelo de GMM se tiene que la probabilidad de una muestra, dado nuestro modelo de mezcla de gaussianas es:

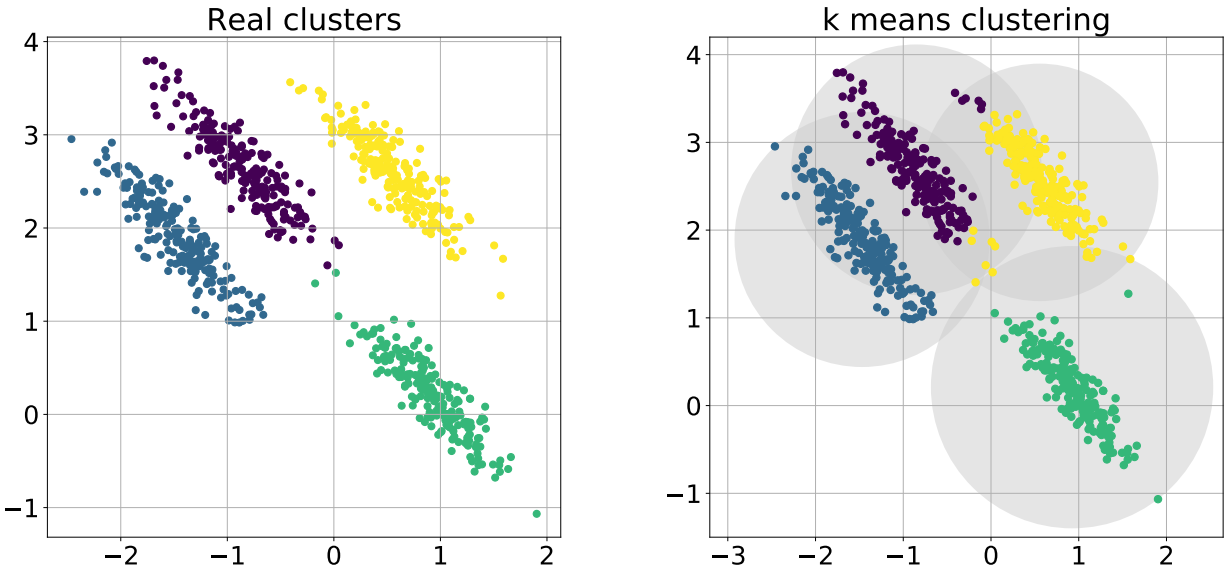
$$p(x_i|\theta) = \sum_{k=1} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k) \tag{270}$$

donde:

$$\mathcal{N}(x_i|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(x_i - \mu_k)^T \Sigma_k^{-1}(x_i - \mu_k)\right] \tag{271}$$

y π_k son los pesos de las mezclas. La forma de encontrar los parámetros es la misma que en k-means, solo que esta vez en el paso de maximización, se debe estimar μ_k, Σ_k y π_k.

Esto se resume como:



(a) (b)

Figura 32: (a) Datos reales con sus etiquetas correctas. (b) Clusters encontrados por k-means.

- **E-step:** Evaluamos las probabilidades posteriores:

$$r_{ik} = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{k'=1}^{K} \pi_{k'} \mathcal{N}(x_i|\mu_{k'}, \Sigma_{k'})} \tag{272}$$

- **M-step:** Se re-estiman los parámetros, usando:

$$\mu_k^{new} = \frac{1}{R_k} \sum_{i=1}^{N} r_{ik} x_i \tag{273}$$

$$\Sigma_k^{new} = \frac{1}{R_k} \sum_{i=1}^{N} r_{ik} (x_i - \mu_k^{new})(x_i - \mu_k^{new})^T \tag{274}$$

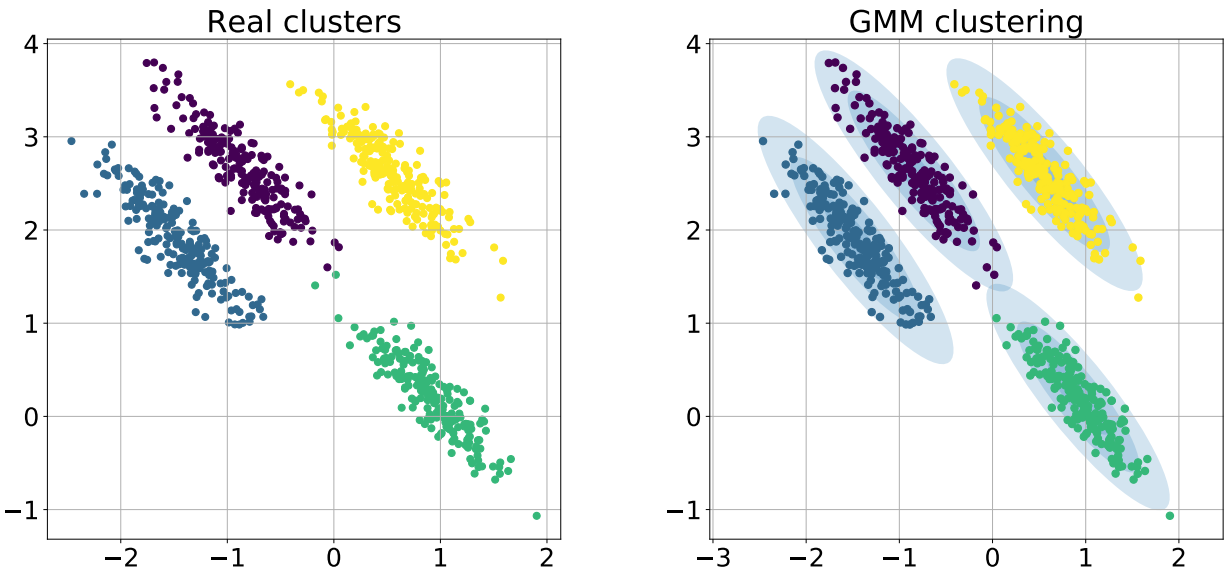
$$\pi_k^{new} = \frac{R_k}{R} \tag{275}$$

Donde

$$R_k = \sum_{i=1}^{N} r_{ik} \qquad \text{y} \qquad R = \sum_{k=1}^{K} R_k.$$

El criterio de detención es evaluar la función de verosimilitud hasta que converja.

Ejemplo: La figura 33 muestra un ejemplo de clustering utilizando GMM. En este caso, se puede observar directamente como GMM es una generalización de kmeans, en donde ahora los clusters tienen forma de gaussiana anisotrópica.



(a) (b)

Figura 33: (a) Datos reales con sus etiquetas correctas. (b) Clusters encontrados por GMM.

9.3. Density-based spatial clustering of applications with noise (DBSCAN)

Es un algoritmo de clustering propuesto por Martin Ester et al. el cual ha tenido mucha popularidad puesto que no requiere definir una cantidad inicial de número de clusters. Los hiperparámetros de entrada del modelo son 2:

- Mínimo número de punto $minPts$.
- Radio o vecindad ϵ.

El algoritmo se basa en la idea de que dado 2 púntos x_i, x_j dentro de un mismo cluster, se dice que x_i *es alcanzable por* x_j, si siempre se puede llegar de x_i a x_j avanzando de punto en punto, donde la distancia entre cada uno es al menos menor que ϵ y además un punto intermedio es un punto núcleo. Con estos el algoritmo define tres tipos de puntos:

- **Puntos núcleo:** Son puntos x_i tales que en una vecindad ϵ tienen almenos $minPts$ vecinos.
- **Puntos borde:** Constituyen el *borde externo* de los cluster.
- **Outliers:** Puntos que no son alcanzables por ningún punto.

Notemos que con lo anterior, se desprende que todo cluster debe tener al menos un punto núcleo. El algoritmo para encontrar los clusters es el siguiente:

Ejemplo: La figura 34 muestra un ejemplo de clustering utilizando DBSCAN. La figura 34(b) muestra en negro los puntos que son clasificados como ruido o *outliers* por el algoritmo. Por otro lado, los puntos núcleos son graficados como un punto grande, mientras que los puntos borde se grafican con un marcador pequeño.

Algoritmo 3 Pseudo código de DBSCAN

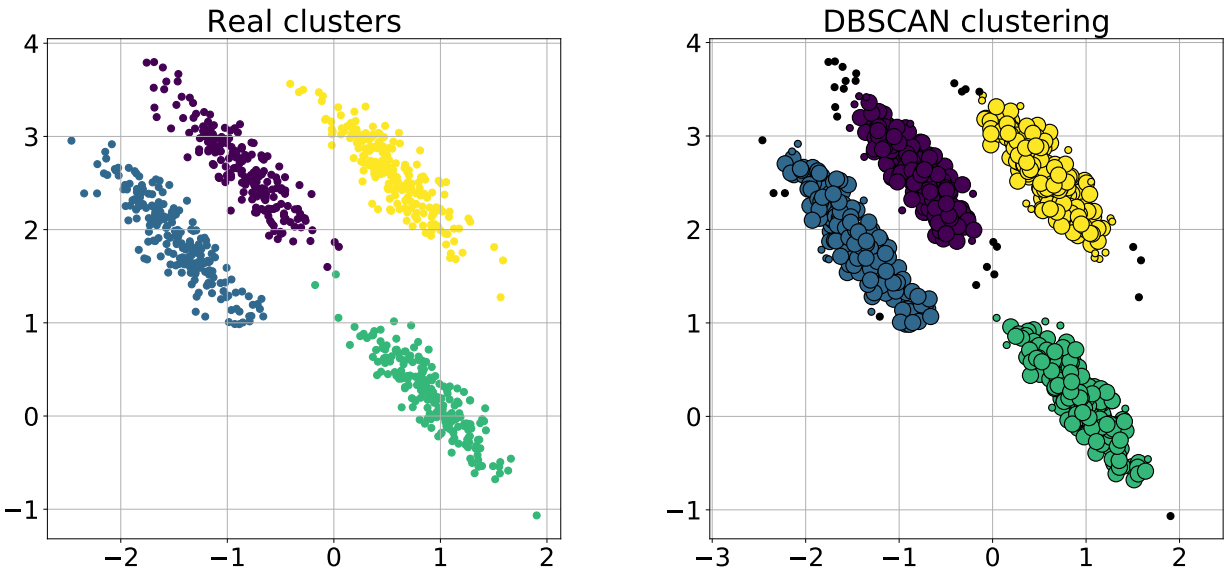
```
1: function DBSCAN(D, eps, MinPts)
2:     C ← 0
3:     for cada punto P no visitado en D do
4:         marcar P como visitado
5:         if sizeOf(PuntosVecinos) ≤ MinPts then
6:             marcar P como RUIDO
7:         else
8:             C ← C+1
9:             expandirCluster(P,vecinos, C, eps, MinPts)
```

Algoritmo 4 Función para expandir cluster.

```
1: function expandirCluster(P, vecinosPts, C, eps, MinPts)
2:     agregar P al cluster C
3:     for cada punto P' en vecinosPts do
4:         if P' no fue visitado then
5:             marcar P' como visitado
6:             vecinosPts' ← regionDeConsulta(P', eps)
7:             if sizeof(vecinosPts') ≥ MinPts then
8:                 vecinosPts ← vecinosPts ∪ vecinosPts'
9:         if P' no tiene cluster asignado then
10:            P' se le asigna el cluster C
```

Algoritmo 5 Retorna los puntos de la vecindad de búsqueda para un punto.

```
1: function regionDeConsulta(P, eps)
       return Todos los puntos junto a P' que están a eps de distancia (incluyendo P)
```



(a) (b)

Figura 34: (a) Datos reales con sus etiquetas correctas. (b) Clusters encontrados por DBSCAN.