UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Matemática

Tesis de Licenciatura

Detección de errores mediante modelos de Deep Learning y Autoencoders aplicado a datos meteorológicos

León Maislín

Director: Andrés Farall

Fecha de Presentación: 11/08/22

Agradecimientos

Fueron varios años de carrera y hay muchas personas a las que quiero agradecer por haberme acompañado en los distintos momentos de este trayecto.

Antes que nada quiero agradecerle a Andrés mi director por haberme acompañado en este proceso, me eseñó y ayudo muchísimo, también es un excelente docente, no pierdan la oportunidad de cursar con el.

A Grabiela, mi profesora de Matemática del secundario, una excelente profesora cuyas enseñansas fueron muy útiles cuando comencé el CBC.

A los grandes profesores de exactas que tuve durante todos los años de cursada, Daniel Galicer, Felipe Marceca, Willy Durán, Daniel Carando, Agustín Gravano, Diego Rial, Mariela Zued.

A Julián Bonder y Matías López, por haber aceptado ser los jurados de esta tesis y tomarse el tiempo de leerla.

A compañeros y amigos que hice durante la cursada, Andy, Juany, Nico, Nazareno.

Al grupo de los Bolzano, Agu, Flor, Alan, Chanchu, Gus, Gordo, Pierna con quienes compartí muchas cursadas, juntadas y risas durante todos estos años y que aún hoy, tengo la suerte de poder seguir compartiendo mis nuevas etapas junto a ellos.

A Agu, mi gran compañera de estudio, con quien hice la mayor parte de la carrera, fue mi gran apoyo y sostén durante esta larga etapa, a quien siempre le voy a estar agradecido ya que sin su ayuda no se si hubiera podido llegar hasta el final.

A mis amigos de Salto de toda la vida, por estar siempre apoyándome, tratando de entender que significa estudiar matemática y siempre estar dispuestos a ayudarme cuando necesitaba relajarme.

A mi abuela, con quien comencé esta etapa de la matemática, ya que desde chico siempre hacía ejercicios en su casa y aunque no pudo llegar al final, sé que donde sea que esté, siempre me va a acompañar.

Por último quiero agradecer a toda mi familia por el apoyo incondicional que siempre me brindaron y en especial a mi mamá y mi papá ya que sin ellos nada de esto hubiera sido posible. Al gran esfuerzo que hicieron para que yo pudiera concentrarme solo en la carrera. Por los valores que me inculcaron durante todos estos años gracias a los cuales, me ayudaron y me ayudan a ser una mejor persona cada día.

Esta Tesis quiero dedicársela a ustedes dos

Índice

1. Introducción

Hoy en día, la inteligencia artificial (IA) es un campo próspero con muchas aplicaciones prácticas y temas de investigación activos. Buscamos software inteligentes para automatizar el trabajo de rutina, comprender el habla o las imágenes, hacer diagnósticos en medicina y apoyar la investigación científica básica.

En los primeros días de la inteligencia artificial, el campo abordaba y resolvía rápidamente problemas que son intelectualmente difíciles para los seres humanos pero relativamente sencillos para computadoras: problemas que pueden describirse mediante una lista de reglas matemáticas formales. El verdadero desafío para la inteligencia artificial resultó ser resolver las tareas que son fáciles de realizar para las personas pero difíciles de describir formalmente: problemas que resolvemos intuitivamente, que se sienten automáticos, como reconocer palabras habladas o rostros en imágenes.

La solución para esto, es permitir que las computadoras aprendan de la experiencia y comprendan el mundo en términos de una jerarquía de conceptos, con cada concepto definido a través de una relación con conceptos más simples. Al recopilar conocimiento de la experiencia, este enfoque evita la necesidad de que los operadores humanos especifiquen formalmente todo el conocimiento que necesita la computadora. La jerarquía de conceptos permite que la computadora aprenda conceptos complicados al construirlos a partir de otros más simples. Si dibujamos un gráfico que muestre cómo estos conceptos se construyen uno encima del otro, el gráfico es profundo, con muchas capas. Por esta razón, llamamos a este enfoque de la IA aprendizaje profundo, conocido como **Deep Learning**.

Lo que buscamos con esta tesis es proponer un modelo estadístico para el problema de detección de errores en datos meteorológicos basado en autoencoders. Primero utilizaremos un conjunto de datos que nosotros creamos, donde podremos ver los gráficos y entender que es lo que hace nuestro modelo, luego utilizaremos lo que aprendimos para poder aplicarlo a un conjunto de datos reales.

1.1. La arquitectura básica de las redes neuronales

En esta sección, presentaremos las redes neuronales de una y varias capas. En la red de una sola capa, a un conjunto de entrada(inputs) se asigna directamente a una salida(output) mediante el uso de una variación generalizada de una función lineal. Esta simple instanciación de una red neuronal también se conoce como perceptrón podemos ver su arquitectura en la Figura 1. En las redes neuronales multicapa, las neuronas están dispuestas en capas, en cuyas capas de entrada y salida están separadas por un grupo de capas ocultas. Esta arquitectura de capa de aprendizaje de la red neuronal también se la conoce como red de avance. En esta sección analizaremos las redes de una sola capa y de varias capas

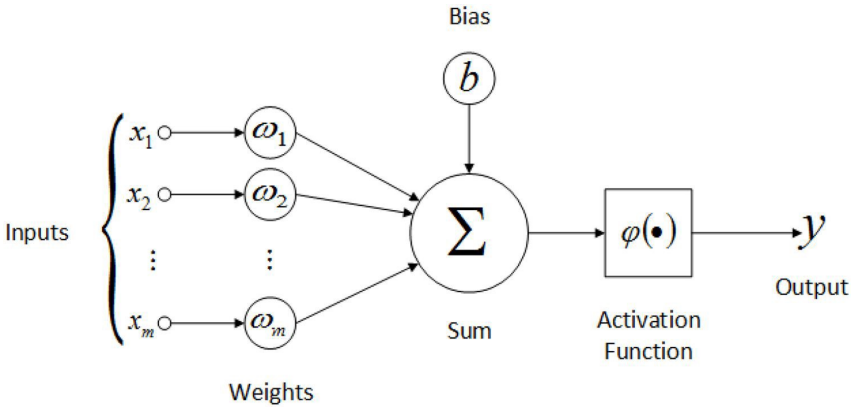


Figura 1: Arquitectura básica de un perceptrón

1.1.1. Única capa computacional: El percetrón

La red neuronal más simple se conoce como perceptrón. Esta red neuronal contiene una sola capa de entrada y un nodo de salida. En la figura 1 se ve la arquitectura básica del perceptrón. Consideremos una situación en la que cada instancia de entrenamiento tiene la forma ($\overline{X}$, y), donde cada $\overline{X} = [x_1, ...x_d]$ contiene d variables de características, $y \in \{+1, -1\}$ contiene el valor observado de la variable de clase binaria. Por "valor observado"nos referimos al hecho de que se nos proporciona como parte de los datos de entrenamiento, y nuestro objetivo es predecir la variable de clase para casos en los que no se observa. Por ejemplo, en una aplicación de detección de fraude con tarjetas de crédito, las características pueden representar varias propiedades de un conjunto de transacciones de tarjetas de crédito (por ejemplo, cantidad y frecuencia de las transacciones), y la variable de clase podría representar Sí o No este conjunto de transacciones es fraudulento. Claramente, en este tipo de aplicación, uno tendría casos históricos en los que se observa la variable de clase, y otros casos (actuales) en los que la variable de clase aún no se ha observado, pero debe predecirse. La capa de entrada contiene d nodos que transmiten las d características $\overline{X} = [x_1, ...x_d]$ con bordes de peso $\overline{W} = [w_1, ...w_d]$ a un nodo de salida. La capa de entrada no lleva a cabo ningún cálculo por derecho propio. La función lineal $\overline{W} \cdot \overline{X} = \sum_{i=1}^{d} w_i x_i$ se computa como el nodo de salida. Posteriormente, se utiliza el signo de este valor real para predecir la variable dependiente de $\overline{X}$. Por lo tanto, la predicción $\hat{y}$ se calcula de la siguiente manera:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\} \tag{1.1}$$

La función de signo asigna un valor real a +1 o -1, lo que es apropiado para la clasificación binaria. Tengamos en cuenta el circunflejo en la parte superior de la variable y para indicar que es un valor predicho en lugar de un valor observado. El error de la predicción es, por tanto, $E(\overline{X}) = y - \hat{y}$, que es uno de los valores extraídos del conjunto {- 2, 0, +2}. En los casos en los que el valor de error $E(\overline{X})$ es distinto de cero, los pesos en la red neuronal deben actualizarse en la dirección (negativa) del gradiente de error. A pesar de la similitud del perceptrón con respecto a los modelos tradicionales de aprendizaje automático, su interpretación como unidad computacional es muy útil porque nos permite juntar múltiples unidades para crear modelos mucho más potentes que los disponibles en el aprendizaje automático tradicional.

La arquitectura del perceptrón se muestra en la figura 1, en la que una sola capa de entrada transmite las características al nodo de salida. Los bordes de la entrada a la salida contienen los pesos $w_1, ...w_d$ con los que las características se multiplican y agregan en el nodo de salida. Posteriormente, se aplica la función de *sign* para convertir el valor agregado en una etiqueta de clase. La función signo cumple el papel de una función de activación. Diferentes opciones de las funciones de activación se puede utilizar para simular diferentes tipos de modelos utilizados en el aprendizaje automático, como *regresión de mínimos cuadrados con objetivos numéricos*, *support vector machine*, *o un clasificador de regresión logística*. La mayoría de los modelos básicos de aprendizaje automático se pueden representar como arquitecturas de redes neuronales simples. Es un ejercicio útil para modelar las técnicas de aprendizaje automático como arquitecturas neuronales, porque proporciona una imagen más clara de cómo el aprendizaje profundo generaliza el aprendizaje automático tradicional. Cabe señalar que el perceptrón contiene dos capas, aunque la capa de entrada no realiza ningún cálculo y solo transmite los valores de las características. La capa de entrada no se incluye en el recuento del número de capas en una red neuronal. Dado que el perceptrón contiene una sola capa *computacional*, se considera una capa única.

En muchos entornos, hay una parte invariante de la predicción, que se conoce como el sesgo. Por ejemplo, considere un entorno en el que las características de las variables están centradas en la media, pero la media de la predicción de la clase binaria de $\{-1, +1\}$ no es 0. Esto tenderá a ocurrir en situaciones en las que la distribución de clases binarias esten muy desequilibrada. En ese caso, el enfoque antes mencionado no es suficiente para la predicción. Necesitamos incorporar un variable de sesgo adicional b que captura esta parte invariante de la predicción:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X} + b\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j + b\} \tag{1.2}$$

El sesgo se puede incorporar como el peso de un borde utilizando una neurona de sesgo. Este es logrado agregando una neurona que siempre transmite un valor de 1 al nodo de salida. Los pesos del borde que conecta la neurona de polarización con el nodo de salida proporciona la variable de polarización. En la figura 1 se muestra un ejemplo de neurona de sesgo. Otro enfoque que funciona bien con arquitecturas de una sola capa es utilizar un truco de ingeniería de características en el que se crea una característica adicional con un valor constante de 1. El coeficiente de esta característica proporciona el sesgo, y luego se puede trabajar con la ecuación 1.1

En el momento en que Rosenblatt [1] propuso el algoritmo de perceptrón, estas optimizaciones se realizaron de forma heurística con circuitos de hardware reales, y no presentado en términos de una noción formal de optimización en el aprendizaje automático (como es común hoy dia). Sin embargo, el objetivo siempre fue minimizar el error en la predicción, incluso si no se presentó una formulación de optimización formal. El algoritmo del perceptrón fue, por tanto, diseñado heurísticamente para minimizar el número de clasificaciones erróneas y había pruebas de convergencia que proporcionaban garantías de corrección del algoritmo de aprendizaje en simplificados ajustes. Por lo tanto, todavía podemos escribir el objetivo (motivado heurísticamente) del algoritmo del perceptrón en forma de mínimos cuadrados con respecto a todas las instancias de entrenamiento en un conjunto de datos $\mathfrak{D}$ que contiene los pares caracteristica-etiqueta:

$$\text{Minimize}_{\overline{W}} L = \sum_{(\overline{X},y)\in\mathfrak{D}} (y - \hat{y})^2 = \sum_{(\overline{X},y)\in\mathfrak{D}} (y - \text{sign}\{\overline{W} \cdot \overline{X}\})^2$$

Este tipo de función objetivo de minimización también se conoce como función de pérdida. Casi todos los algoritmos de aprendizaje de redes neuronales se formulan con el uso de una función de pérdida. Esta función de pérdida se parece mucho a la regresión de mínimos cuadrados. Sin embargo, este último se define para las variables objetivo de valor continuo, y la pérdida correspondiente es una función suave y continua de las variables. Por otro lado, para la forma de la función objetivo de mínimos cuadrados, la función de signo no es diferenciable, con saltos escalonados en puntos específicos. Además, la función de signo toma valores constantes en grandes porciones del dominio y, por lo tanto, el gradiente exacto toma el valor cero en puntos diferenciables. Esto da como resultado una superficie de pérdida similar a una escalera, que no es adecuado para el descenso de gradientes. El algoritmo de perceptrón (implícitamente) utiliza una suave aproximación del gradiente de esta función objetivo con respecto a cada ejemplo:

$$\nabla L_{smooth} = \sum_{(\overline{X},y)\in\mathfrak{D}} (y - \hat{y})\overline{X} \tag{1.3}$$

Tengamos en cuenta que el gradiente anterior no es un verdadero gradiente de la superficie en forma de escalera de la función objetivo (heurística), que no proporciona gradientes útiles. Por tanto, la escalera es suavizada en una superficie inclinada definida por el *criterio del perceptrón* . Cabe destacar que conceptos como el "criterio del perceptrón"se propusieron más tarde que el artículo original de Rosenblatt [1] para explicar los pasos heurísticos de descenso de gradientes. Por ahora, asumiremos que el algoritmo del perceptrón optimiza alguna función suave desconocida con el uso del descenso del gradiente. Aunque la función objetivo anterior se define sobre todos los datos de entrenamiento, el algoritmo de entrenamiento de las redes neuronales funciona alimentando cada instancia de datos de entrada $\overline{X}$ dentro de la red uno por uno (o en pequeños baches) para crear la predicción . Los pesos son entonces actualizado, basado en el valor de error $E(\overline{X}) = (y - \hat{y})$. Específicamente, cuando los puntos de datos $\overline{X}$ son alimentados a la red, el vector de peso $\overline{W}$ se actualiza de la siguiente manera:

$$\overline{W} \Leftarrow \overline{W} + \alpha(y - \hat{y})\overline{X} \tag{1.4}$$

El parámetro α regula la tasa de aprendizaje de la red neuronal. El algoritmo del perceptrón recorre repetidamente todos los ejemplos de entrenamiento en orden aleatorio y ajusta iterativamente los pesos hasta que se alcance la convergencia. Se puede recorrer un único punto de datos de entrenamiento muchas

veces. Cada uno de estos ciclos se conoce como *epoch*. También se puede escribir la actualización gradiente descente en términos del error $E(\overline{X}) = (y - \hat{y})$. de la siguiente manera:

$$\overline{W} \Leftarrow \overline{W} + \alpha E(\overline{X})\overline{X} \tag{1.5}$$

El algoritmo básico del perceptrón puede considerarse un método de descenso de gradiente estocástico, que minimiza implícitamente el error al cuadrado de la predicción, al realizar un descenso de gradiente con actualizaciones respecto a los puntos de entrenamiento elegidos al azar. La suposición es que la red neuronal recorre los puntos en orden aleatorio durante el entrenamiento y cambia los pesos con el objetivo de reducir el error de predicción en ese punto. Es fácil ver que en la Ecuación 1.5 se realizan actualizaciones distintas de cero a las ponderaciones solo cuando $y \neq \hat{y}$, lo cual ocurre solo cuando se cometen errores en la predicción. En el *descenso de gradiente estocástico con mini baches* , las actualizaciones de la Ecuación 1.5 antes mencionadas se implementan en un subconjunto de entrenamiento elegido al azar puntos S:

$$\overline{W} \Leftarrow \overline{W} + \alpha \sum_{\overline{X} \in S} E(\overline{X})\overline{X} \tag{1.6}$$

Una peculiaridad interesante del perceptrón es que es posible establecer la taza de aprendizaje α a 1, porque en la tasa de aprendizaje solo escalan los pesos. El tipo de modelo propuesto en el perceptrón es un modelo lineal, en el que la ecuación $\overline{W} \cdot \overline{X} = 0$ define un hiperplano lineal. Aquí, $\overline{W} = (w_1 ... w_d)$ es un vector d-dimensional que es normal al hiperplano. Además, el valor de $\overline{W} \cdot \overline{X}$ es positivo para los valores de $\overline{X}$ en un lado del hiperplano, y es negativo para los valores de $\overline{X}$ en el otro lado. Este tipo de modelo funciona particularmente bien cuando los datos son *linealmente separable*. Ejemplos de linealmente separable e inseparables se muestran en la Figuras 2 y 3.

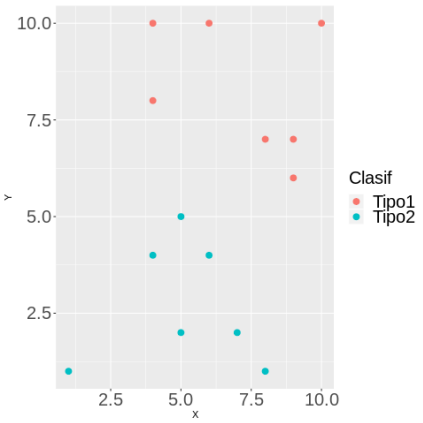


Figura 2: Datos linealmente separables

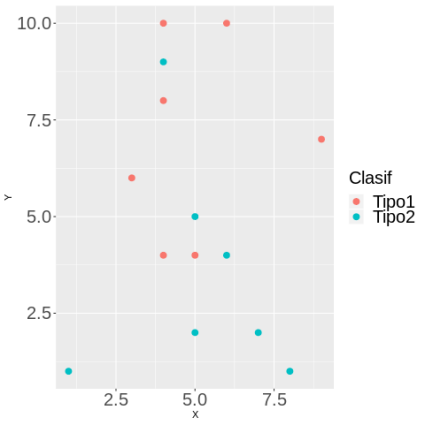


Figura 3: Datos no separables linealmente

El algoritmo de perceptrón es bueno para clasificar conjuntos de datos como el que se muestra del lado izquierdo, la Figura 2, cuando los datos son linealmente separables. Por otro lado, tiende a tener un desempeño deficiente en conjuntos de datos como el que se muestra en el lado derecho en la Figura 3. Este ejemplo muestra la limitación de modelado inherente de un perceptrón, que requiere el uso de arquitecturas neuronales más complejas. Dado que el algoritmo del perceptrón original se propuso como una minimización heurística de errores de clasificación, era particularmente importante mostrar que el algoritmo converge a soluciones razonables en algunos casos especiales. En este contexto, se demostró [1] que el algoritmo del perceptrón siempre converge para proporcionar un error cero en los datos de entrenamiento cuando los datos son linealmente separables. Sin embargo, no se garantiza que el algoritmo de perceptrón convergen en casos en los que los datos no son separables linealmente. El perceptrón a veces puede llegar a una solución muy pobre con datos que no son linealmente separables (en comparación con muchos otros algoritmos de aprendizaje).

1.1.2. ¿Qué función objetivo está optimizando el perceptrón?

Como se discutió anteriormente, el artículo de perceptrón original de Rosenblatt [1] no se propone formalmente una función de pérdida. En esos años, estas implementaciones se lograron utilizando circuitos de hardware reales. El *perceptrón Mark I* original estaba destinado a ser una máquina en lugar de un algoritmo, y se usó un hardware personalizado para crearlo (ver Figura 4)

Figura 4: El algoritmo del perceptrón se implementó originalmente utilizando circuitos de hardware. La imagen muestra la máquina perceptrón Mark I construida en 1958

El objetivo general era minimizar el número de errores de clasificación con un proceso de actualización heurística (en hardware) que cambiaba los pesos en la dirección "correcta" siempre que se producían errores. Esta actualización heurística se parecía mucho al descenso de gradiente, pero no era un derivado del método de descenso por el gradiente. El descenso de gradiente se define solo para funciones de pérdida suave en entornos algorítmicos, mientras que el enfoque centrado en el hardware se diseñó de una manera más heurística con salidas binarias. Muchos de los principios binarios y centrados en circuitos fueron heredados del modelo McCulloch-Pitts [2] de la neurona. Desafortunadamente, las señales binarias no son propensos a la optimización continua. ¿Podemos encontrar una función de pérdida suave, cuyo gradiente resulte ser la actualización del perceptrón? El número de errores de clasificación en un problema de clasificación binaria que se puede escribir en forma de una función de pérdida 0/1 para el punto de datos de entrenamiento $(\overline{X_i}, y_i)$ de la siguiente manera:

$$L_i^{(0/1)} = \frac{1}{2}(y_i - \text{sign}\,\{\overline{W} \cdot \overline{X_i}\})^2 = 1 - y_i\,\text{sign}\,\{\overline{W} \cdot \overline{X_i}\} \tag{1.7}$$

La simplificación del lado derecho de la función objetivo anterior se obtiene dando el valor a y_i^2 y $\text{sign}\,\{\overline{W} \cdot \overline{X_i}\}^2$ de 1, ya que se obtienen elevando al cuadrado un valor extraído de $\{-1, +1\}$. Sin embargo, esta función objetivo no es diferenciable, porque tiene una forma similar a una escalera, especialmente cuando se agrega sobre múltiples puntos. Tengamos en cuenta que la pérdida 0/1 anterior está dominado por el término $-y_i\,\text{sign}\,\{\overline{W} \cdot \overline{X_i}\}$, en el que la función de signo causa la mayor parte de los problemas asociados con la no diferenciabilidad. Dado que las redes neuronales se definen por optimización basada en gradientes, necesitamos definir una función objetivo suave que sea responsable de las actualizaciones del perceptrón. Se puede demostrar [3] que las actualizaciones del perceptrón optimizan implícitamente el *criterio del perceptrón* . Esta función objetivo se define eliminando la función de signo en la pérdida 0/1 anterior y establecer valores negativos en 0 para tratar todas las predicciones correctas de forma uniforme y sin pérdidas:

$$L_i = max\{-y_i(\overline{W} \cdot \overline{X_i}), 0\} \tag{1.8}$$

El gradiente de esta función objetivo suavizada conduce a la actualización del perceptrón, y la actualiza-

ción del perceptrón es esencialmente $\overline{W} \Leftarrow \overline{W} - \alpha \nabla_W L_i$. La función de pérdida modificada para permitir el cálculo del gradiente de una función no diferenciable también se denomina *función de pérdida sustituta suavizada*. Casi todos métodos de aprendizaje basados en optimización continua (como redes neuronales) con salidas discretas (como las etiquetas de clase) utilizan algún tipo de función de pérdida sustituta suavizada.

Aunque el criterio de perceptrón mencionado anteriormente fue modificado mediante ingeniería inversa trabajando con actualizaciones del perceptrón al revés, la naturaleza de esta función de pérdida expone algunas de las debilidades de las actualizaciones en el algoritmo original. Una observación interesante sobre el criterio del perceptrón es que se puede establecer $\overline{W}$ en el vector cero *independientemente de los datos de entrenamiento* establecido para obtener el valor de pérdida óptimo de 0. A pesar de este hecho, la actualización del perceptrón continua convergiendo a un claro separador entre las dos clases en casos linealmente separables; después de todo, un separador entre las dos clases también proporciona un valor de pérdida de 0. Sin embargo, el comportamiento de los datos que no son linealmente separables es bastante arbitrario, y la solución resultante a veces ni siquiera es un buen separador aproximado de las clases. La sensibilidad directa de la pérdida a la *magnitud* del vector de peso puede diluir el objetivo de la separación de clases; es posible que las actualizaciones empeoren significativamente el número de clasificaciones erróneas mientras mejora la pérdida. Este es un ejemplo de cómo las funciones de pérdida sustituta pueden a veces no alcanzar plenamente sus objetivos previstos. Debido a este hecho, el enfoque no es estable y puede producir soluciones de calidad muy variable.

Por lo tanto, se propusieron varias variaciones del algoritmo de aprendizaje para datos inseparables y un enfoque natural es siempre realizar un seguimiento de la mejor solución en términos del número de clasificaciones erróneas [4]. Este enfoque de mantener siempre la mejor solución en el "bolsillo"se conoce como el *algoritmo de bolsillo* . Otra variante de alto rendimiento incorpora la noción de margen en la función de pérdida, que crea un algoritmo *idéntico* al *lineal support vector machine* . Por esta razón, el lineal support vector machine también se denomina como *el perceptrón de estabilidad óptima* .

1.1.3. Elección de funciones de activación y pérdida

La elección de la función de activación es una parte fundamental del diseño de una red neuronal. En el caso del perceptrón, la elección de la función de activación del signo está motivada por el hecho de predecir una clase binaria. Sin embargo, es posible tener otro tipo de situaciones donde se pueden predecir diferentes variables objetivo. Por ejemplo, si la variable a predecir es del tipo real, entonces tiene sentido usar la función de activación identidad, y el algoritmo resultante es el mismo que el de la regresión por mínimos cuadrados. Si queremos predecir una probabilidad de una clase binaria, tiene sentido usar una función sigmoidea para activar el nodo de salida, por lo que que la predicción $\hat{y}$ indica la probabilidad de que el valor observado, y, de la variable dependiente sea 1

La importancia de las funciones de activación no lineales se vuelve más significativa cuando uno cambia de el perceptrón de una sola capa a las arquitecturas de múltiples capas. Existen diferentes tipos de funciones no lineales como el *signo* , *sigmoideas* o *tangentes hiperbólicas* que se pueden usar cuando uno tiene varias capas. Usamos la notación ϕ para denotar la función de activación:

$$\hat{y} = \phi(\overline{W} \cdot \overline{X}) \tag{1.9}$$

Por lo tanto, una neurona realmente calcula dos funciones dentro del nodo, razón por la cual se incorporó el símbolo de suma $\sum$ así como el símbolo de activación ϕ dentro de una neurona. La división de estos cálculos de neuronas en dos valores separados se muestra en la Figura 5

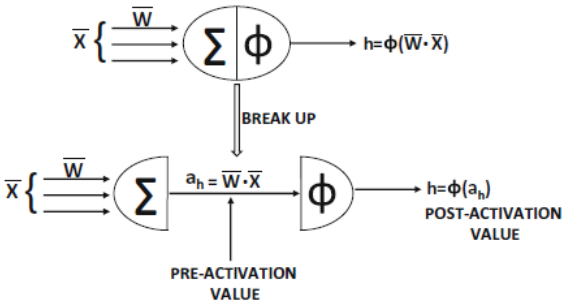


Figura 5: Valores de pre-activación y pos-activación dentro de una neurona

El valor calculado antes de aplicar la función de activación $\phi(\cdot)$ lo denominaremos valor de pre-activación, mientras que el valor calculado después de aplicar la función de activación es denominado valor pos-activación. La salida de una neurona es siempre el valor de post-activación, aunque las variables de pre-activación se utilizan a menudo en diferentes tipos de análisis, como los cálculos del *algoritmo de backpropagation*.

La función de activación más básica $\phi(\cdot)$ es la identidad o función de activación lineal, que proporciona linealidad: $\phi(v) = v$

La función de activación lineal se utiliza a menudo en el nodo de salida, cuando el objetivo es un valor real. Las funciones de activación clásicas que se utilizaron en los primeros desarrollos de las redes neuronales eran las funciones de *signo*, *sigmoidea* y *tangente hiperbólica*:

$$\phi(v) = \text{sign}(v)(\text{función signo})$$

$$\phi(v) = \frac{1}{1+e^{-v}}(\text{función sigmoidea})$$

$$\phi(v) = \frac{e^{2v}-1}{e^{2v}+1}(\text{función tanh})$$

Si bien la función de activación signo se puede utilizar para mapear las salidas binarias en el momento de la predicción, su no diferenciabilidad impide su uso para crear la función de pérdida en el momento del entrenamiento. La activación sigmoidea genera un valor en $(0, 1)$, que es útil para realizar cálculos que deben interpretarse como probabilidades.

La función tanh tiene una forma similar a la de la función sigmoidea, excepto que se reescala horizontalmente y verticalmente a $[-1, 1]$. Las funciones *tanh* y sigmoidea están relacionadas de la siguiente manera:

$$tanh(v) = 2 \cdot sigmoidea(2v) - 1$$

La función tanh es preferible a la sigmoidea cuando se desea que las salidas de los cálculos sean tanto positivas como negativas. Además, al estar centrada en la media y tener gradiente más grande (debido al estiramiento) con respecto a la sigmoidea hace que sea más fácil de entrenar. Las funciones sigmoidea y la tanh han sido las herramientas históricas elegidas para incorporar la no linealidad en la red neuronal. En los últimos años, sin embargo, una serie de funciones de activación lineal por partes se han vuelto más populares:

$$\phi(v) = max\{v, 0\}(\text{función ReLU(Rectified Linear Unit)})$$

$$\phi(v) = max\{min\{v, 1\}, -1\}(\text{función tanh dura})$$

Las funciones de activación ReLU y tanh dura han reemplazado en gran medida a las funciones de activación sigmoidea y tanh en las redes neuronales modernas debido a la facilidad para entrenar redes neuronales multicapa con estas funciones de activación.

En la Figura 6 se detallan las funciones que mencionamos antereiormente.

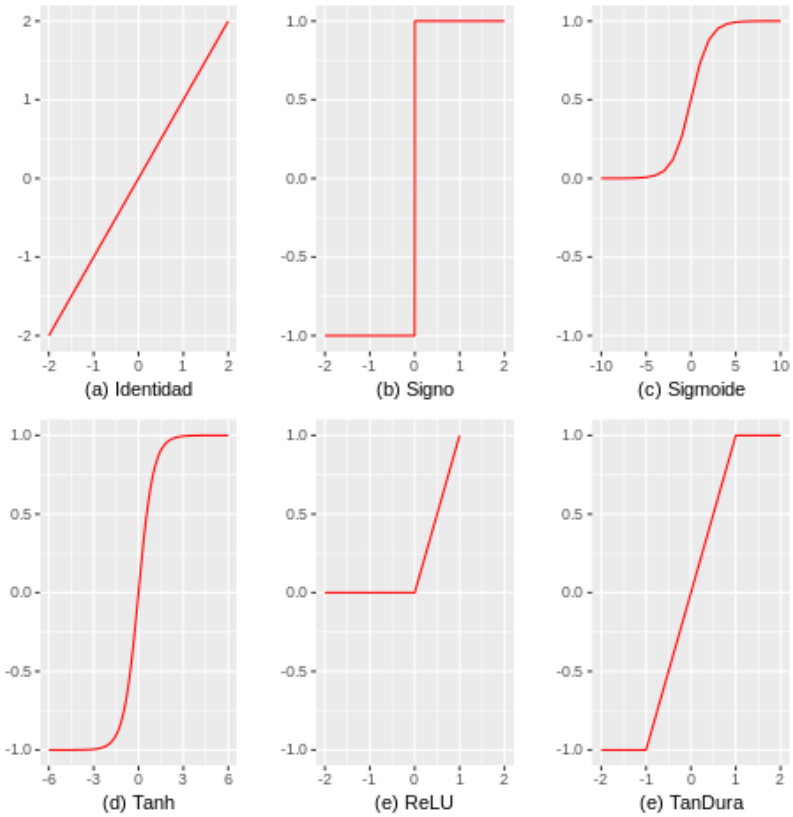


Figura 6: La *Identidad* es una función que a cada elemento de un conjunto lo manda a si mismo. El *Signo* es una función que a todo elemento positivo le asigna el valor 1 y a todo elemento negativo le asigna el valor -1. *Sigmoide* es una función cuyos valores de salidad están entre el rango de 0 y 1 por lo que es interpretada como una probabilidad. La *Tanh* es un escalamiento de la *Sigmoide* que ahora toma valores entre -1 y 1 y además está centrada. *ReLU* esta función es la más utilizada ya que permite un aprendizaje muy rápido en las redes neuronales. La *TanhDura* también es utilizada debido a que las redes neuronales son fáciles de entrenar.

Al igual que la elección de las funciones de activación, la elección de la función de pérdida, juega un papel importante en cualquier modelo estadístico, definen un objetivo contra el cual se evalúa el rendimiento del modelo y los parámetros aprendidos por el modelo se determinan minimizando una función de pérdida elegida. Para modelos donde la variable a predecir es real, como por ejemplo regresiones, las funciones de pérdida más utilizadas son el MSE y el MAE, pero para casos de clasificación es común usar la Cross-Entropy.

- Error cuadrático medio (MAE):

$$\frac{1}{n}\sum_{i=1}^{n}|y^{(i)}-\hat{y}^{(i)}|$$

- Error cuadrático medio (MSE):

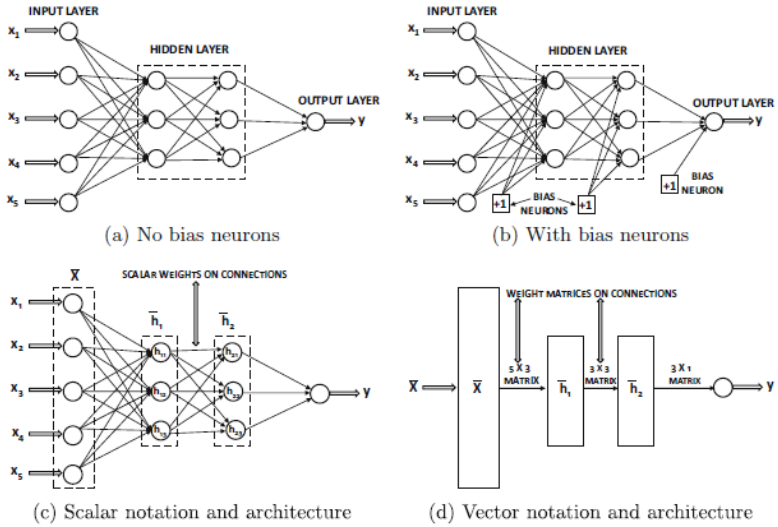
$$\frac{1}{n}\sum_{i=1}^{n}(y^{(i)}-\hat{y}^{(i)})^2$$

- Cross-entropy:

$$\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}log(\hat{y}^{(i)}) + (1-y^{(i)})log(1-\hat{y}^{(i)})]$$

1.2. Redes neuronales multicapa

Las redes neuronales multicapa contienen más de una capa computacional. El perceptrón contiene una capa de entrada y salida, de las cuales la capa de salida es la única capa que realiza cálculos. La capa de entrada transmite los datos a la capa de salida y todos los cálculos son completamente visibles para el usuario. Las redes neuronales multicapa contienen múltiples capas computacionales; las capas intermedias adicionales (entre entrada y salida) son denominadas capas ocultas porque los cálculos realizados no son visibles para el usuario. La arquitectura específica de las redes neuronales multicapa se conoce como redes de alimentación hacia adelante, porque las capas sucesivas se alimentan entre sí en la dirección de avance desde la entrada a la salida. La arquitectura predeterminada de las redes de alimentación directa asume que todos los nodos en una capa están conectados a los de la siguiente capa. Por lo tanto, la arquitectura de la red neural está casi completamente definida una vez que el número de capas y el número/tipo de nodos en cada capa ha sido definida. El único detalle que queda es la función de pérdida que está optimizada en la capa de salida. Aunque el algoritmo del perceptrón utiliza el criterio del perceptrón, este no es la única opción. Es extremadamente común usar salidas softmax con pérdida de entropía cruzada para predicciones discretas y salidas lineales con pérdida al cuadrado para predicción de valor real. Como en el caso de las redes de una sola capa, las neuronas con sesgo se pueden utilizar tanto en las capas ocultas como en las capas de salida. En la Figura 7 (a) y (b) se muestran ejemplos de redes multicapa con o sin neuronas de sesgo respectivamente. En cada caso, la red neuronal contiene tres capas. Tengamos en cuenta que la capa de entrada a menudo no se cuenta, porque simplemente transmite los datos y no se realiza ningún cálculo en esa capa. Si una red neuronal contiene $p_1...p_k$ unidades en cada una de sus k capas, entonces las representaciones vectoriales (de columna) de estas salidas, denotadas por $h_1...h_k$ tienen dimensionalidades $p_1...p_k$. Por lo tanto, el número de unidades en cada capa se conoce como la dimensionalidad de esa capa.



(a) No bias neurons (b) With bias neurons

(c) Scalar notation and architecture (d) Vector notation and architecture

Figura 7: Arquitectura básica de una red feed-forward con dos capas ocultas y una sola capa de salida

Los pesos de las conexiones entre la capa de entrada y la primera capa oculta están contenidos en una matriz W_1 con tamaño $d \times p_1$, mientras que los pesos entre la r-ésima y la (r + 1)-ésima capa oculta se indican mediante la matriz $p_r \times p_{r+1}$ indicada por W_r. Si la capa de salida contiene o nodos, entonces la matriz final W_{k+1} es de tamaño $p_k \times o$. El vector de entrada d-dimensional $\overline{x}$ se transforma en las salidas usando la siguiente recursión de ecuaciones:

$$\overline{h}_1 = \phi(W_1^T \overline{x}) \quad \text{[De la entrada a capa oculta]}$$
$$\overline{h}_{p+1} = \phi(W_{p+1}^T \overline{h}_p) \quad \forall p \in \{1...k-1\} \quad \text{[De capa oculta a capa oculta]}$$
$$\overline{o} = \phi(W_{k+1}^T \overline{h}_k) \quad \text{[De capa oculta a capa de salida]}$$

Aquí, las funciones de activación como la función sigmoidea se aplican en forma de elementos a sus argumentos vectoriales. Sin embargo, algunas funciones de activación como el softmax (que se utilizan normalmente en las capas de salida) naturalmente tienen argumentos vectoriales. Aunque cada unidad de una red neuronal contiene una sola variable, muchos diagramas arquitectónicos combinan las unidades en una sola capa para crear una sola unidad vectorial, que se representa como un rectángulo en lugar de un círculo. Por ejemplo, el diagrama arquitectónico de la figura 7 (c) (con escalar unidades) se ha transformado en una arquitectura neuronal basada en vectores en la figura 7 (d). Notemos que las conexiones entre las unidades vectoriales son ahora matrices. Además, se asume implícitamente que en la arquitectura neuronal basada en vectores, todas las unidades en una capa usan la misma función de activación, que se aplica en forma de elementos a esa capa. Esta restricción no es generalmente un problema, porque la mayoría de las arquitecturas neuronales usan la misma función de activación a lo largo de la tubería computacional, con la única desviación causada por la naturaleza de la capa de salida. Tengamos en cuenta que las ecuaciones de recurrencia y las arquitecturas vectoriales mencionadas anteriormente son válidas solo para redes de alimentación directa por capas, y no siempre se puede utilizar para redes de diseños arquitectónicos no convencionales. Es posible tener todo tipo de diseños no convencionales en los que las entradas pueden incorporarse en capas intermedias o la topología puede permitir conexiones entre capas no consecutivas. Además, las funciones calculadas en un nodo pueden no siempre tener la forma de una combinación de una función lineal y activación. Es posible tener todo tipo de funciones computacionales arbitrarias en los nodos.

Aunque en la figura 7 se muestra un tipo de arquitectura muy clásico, es posible variar en él de muchas maneras, como permitir múltiples nodos de salida. Estas elecciones son a menudo determinado por los objetivos de la aplicación en cuestión (por ejemplo, clasificación o reducción de la dimensión). Un ejemplo clásico de la configuración de reducción de dimensionalidad es el autoencoder, que recrea las salidas de las entradas. Por tanto, el número de salidas y entradas es igual, como se muestra en la figura 8. La capa oculta constreñida en el medio da como resultado una representación reducida de cada instancia. Como resultado de esta constricción, hay alguna pérdida en la representación, que normalmente corresponde al ruido en los datos. Las salidas del las capas ocultas corresponden a la representación reducida de los datos. De hecho, se puede demostrar que una variante poco profunda de este esquema es matemáticamente equivalente a un método de reducción de dimensionalidad bien conocido *descomposición en valores singulares*.

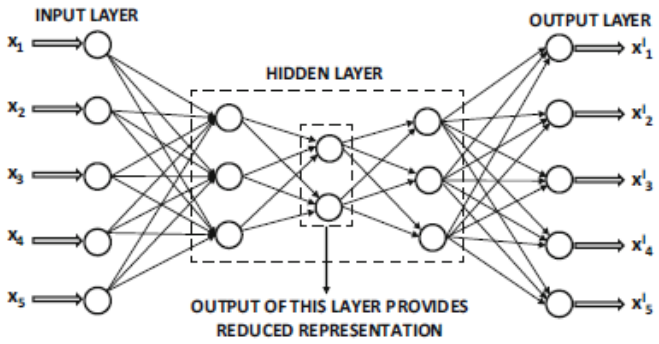


Figura 8: Un ejemplo de un autoencoder con múltiples capas de salida

Aunque una arquitectura completamente conectada puede funcionar bien en muchos entornos, el mejor rendimiento a menudo se logra podando muchas de las conexiones o compartiéndolas en un manera perspicaz. Por lo general, estos conocimientos se obtienen mediante una comprensión de los datos de un dominio específico. Un ejemplo clásico de este tipo de reducción y distribución de peso es el de

la *arquitectura de red neuronal convolucional*,en la que la arquitectura es cuidadosamente diseñada para ajustarse a las propiedades típicas de los datos de imagen. Este enfoque minimiza el riesgo de *sobreajuste* al incorporar conocimientos (o sesgos) específicos del dominio. El sobreajuste es un problema generalizado en el diseño de redes neuronales, de modo que la red suele funcionar muy bien con los datos de entrenamiento, pero se generaliza mal a los datos nuevos que no se han visto (datos de prueba). Este problema se produce cuando el número de los parámetros (que suelen ser iguales al número de conexiones de peso) son demasiado grandes en comparación con el tamaño de los datos de entrenamiento. En tales casos, la gran cantidad de parámetros memorizan los matices específicos de los datos de entrenamiento, pero no reconocer estadísticamente patrones significativos para clasificar los datos de prueba. Claramente, aumentando el número de nodos en la red neuronal tiende a fomentar el sobreajuste. Gran parte del trabajo reciente se ha centrado tanto en la arquitectura de la red neuronal como en los cálculos realizados dentro de cada nodo para minimizar el sobreajuste. Además, la forma en que se entrena la red neuronal también tiene un impacto en la calidad de la solución final. Muchos métodos inteligentes se han propuesto en los últimos años, como el *preentrenamiento* para mejorar la calidad de la solución aprendida.

1.3. Autoencoders

Un autoencoder, es un tipo especial de red neuronal multicapa que realiza una reducción de la dimensión de los datos de forma jerárquica y no lineal. Típicamente, el número de nodos en la capa de entrada(input) es igual al de la capa de salida(output) y la arquitectura es en capas y simétrica. La meta del autoencoder es entrenar el output, para reconstruir el input lo mejor posible. Los nodos en la capa intermedia son en cantidad menores, es a lo que se lo conoce como, cuello de botella o espacio latente, y de esta forma, la única manera de reconstruir el input es aprender pesos para que los outputs intermedios de los nodos en las capas medias puedan hacer representaciones reducidas. Dado que un autoencoder trata de reconstruir el input en lugar de predecir un valor objetivo $\mathbf{Y}$ dado un input $\mathbf{X}$, autoencoders son *modelos de aprendizaje no supervisado.*

Mencionamos que un autoencoder crea una representación reducida de los datos, es así que es un enfoque natural para descubrir errores. La idea básica, es que es mucho más difícil representar correctamente errores que los datos normales. De esta forma, en la reconstrucción de un error, el score va a ser mayor. Esto nos provee una forma natural para clasificar un dato.

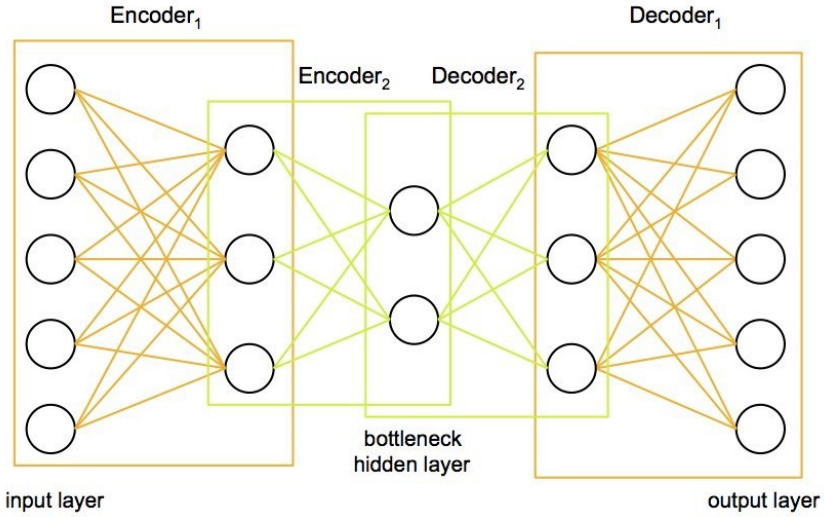


Figura 9: Ejemplo de autoencoder mostrando la parte de codificación y de decodificación

Como vemos en la figura 9, autoencoder siempre consisten de 2 partes, la codificación y la decodificación, que puede ser definidas como funciones σ y ψ tal que:

$\sigma : \mathcal{X} \to \mathcal{F}$

$\psi : \mathcal{F} \to \mathcal{X}$

$\sigma, \psi = \underset{\sigma,\psi}{\arg\min} \|X - (\psi \circ \sigma)X\|^2$ En el caso simple, donde sólo hay una capa oculta, la parte de codificación toma el input $\mathbf{x} \in \mathbb{R}^d = \mathcal{X}$ y lo mapea a $\mathbf{z} \in \mathbb{R}^p = \mathcal{F}$:

$\mathbf{z} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$ Esta imagen $\mathbf{z}$ es usualmente referida como *código, variables latentes, o representación latente*. Donde, ϕ es una función de activación como por ejemplo la función sigmoide o una función lineal, $\mathbf{W}$ es el peso de la matriz y $\mathbf{b}$ es el vector de sesgo. Luego, en la etapa de decodificación el autoencoder mapea $\mathbf{z}$ a una reconstrucción $\mathbf{x}'$ del mismo tamaño que $\mathbf{x}$: $\mathbf{x}' = \phi'(\mathbf{W}'\mathbf{z} + \mathbf{b}')$ donde $\phi', \mathbf{W}',$ y $\mathbf{b}'$ en general para el decodificador van a ser diferentes de los correspondientes $\phi, \mathbf{W},$ y $\mathbf{b}$ para el codificador, dependiendo del diseño del autoencoder.

Los autoencoders también son entrenados para minimizar errores de reconstrucción (como errores cuadráticos):

$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2$ donde $\mathbf{x}$ es usualmente promediado sobre conjunto entrenado de datos de entrada.

Si el espacio latente $\mathcal{F}$ tiene dimensión más pequeña que el espacio de entrada $\mathcal{X}$, entonces el vector de características $\sigma(x)$ puede ser considerado como una representación reducida del dato de entrada x. Si las capas ocultas son más grandes que la capa de entrada, el autoencoder probablemente podría convertirse en la función identidad y volverse inútil.

2. Métodos y herramientas

2.1. Curva ROC

Desde su invención en el seno de las investigaciones militares estadounidenses ha formado parte del Análisis Discriminante y la Teoría de la Detección de Señales. Su primera aplicación fue en detección de señales de radar durante los años 50'. En los 60' Green y Swets la utilizaron para experimentos psicofísicos y más tarde, en los 70', el radiólogo Leo Lusted las usó para decisión de disgnóstico mediante imágenes médicas. A partir de entonces, numerosos investigadores han utilizado ésta herramienta en el campo de la sanidad, la economía, la meteorología y más recientemente en el aprendizaje automático.

La curva ROC es una herramienta estadística utilizada en el análisis de clasificar la capacidad discriminante de una prueba diagnóstica dicotómica. Es decir, una prueba, basada en una variable de decisión, cuyo objetivo es clasificar a los "individuos"de una población en dos grupos: uno que presente un evento de interés y otro que no. En nuestro caso, la población serían los datos meteorológicos, el evento de interés sería clasificar a un dato como un error y la prueba diagnostica sería el modelo de autoencoder que creamos. Esta capacidad discriminante está sujeta al valor umbral elegido de entre todos los posibles resultados de la variable de decisión, es decir, la variable por cuyo resultado se clasifica a cada individuo en un grupo u otro. La curva es el gráfico resultante de representar, para cada valor umbral, las medidas de sensibilidad y especificidad de la prueba diagnóstica. Por un lado, la sensibilidad cuantifica la proporción de individuos que presenta el evento de interés y que son clasificados por la prueba como portadores de dicho evento, es la tasa de verdaderos positivos TPR. Por otro lado, la especificidad cuantifica la proporción de individuos que no lo presentan y son clasificados por la prueba como tal, es $1-$ FPR , donde FPR es la tasa de falsos positivos.

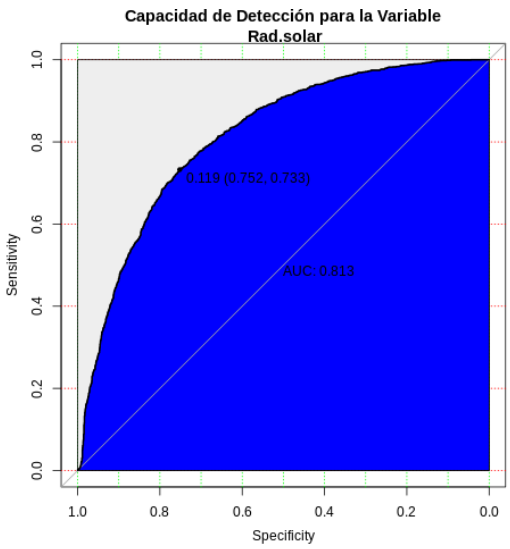


Figura 10: Curva ROC del MSE para la variable Radiasión Solar

2.2. Gráficos violín

Los gráficos violín son similares a los box plot excepto que también muestra la distribución de probabilidad de los datos, generalmente suavizado por un estimador de densidad por núcleos. Este tipo de gráficos aunque son más informativos que los box plot, son menos populares. La diferencia es particularmente útil cuando la distribución de los datos es multimodal, ya que en un box plot no tendríamos forma de visualizarlo.

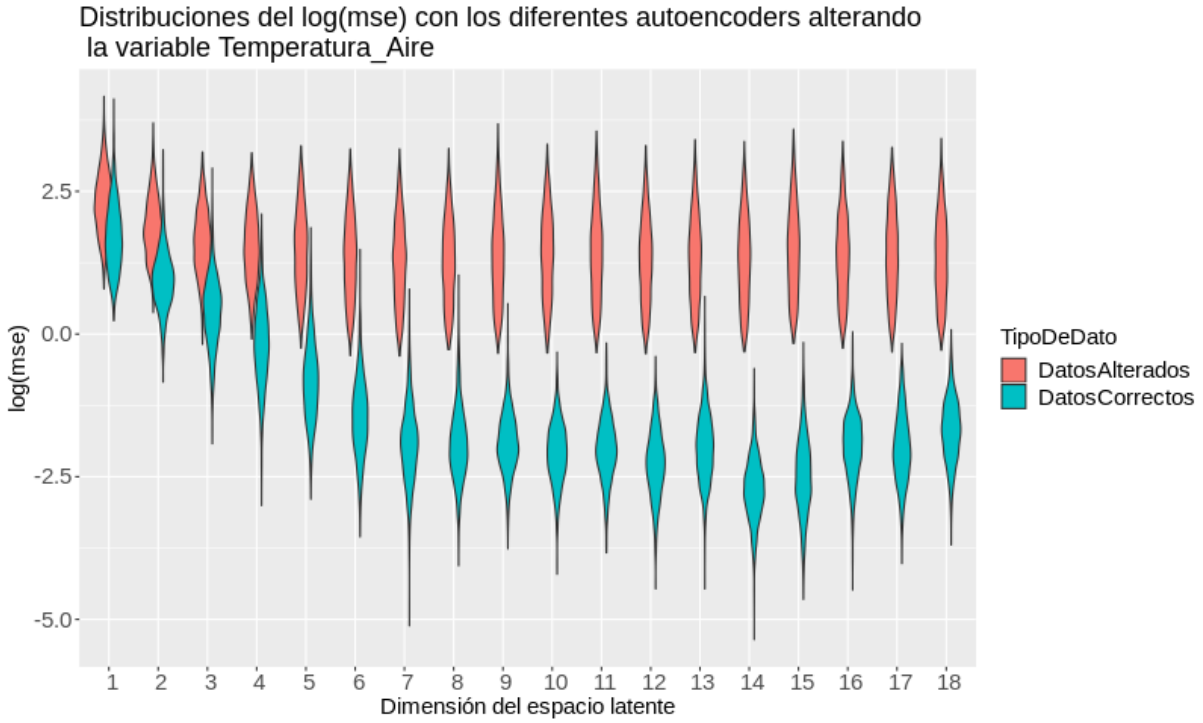


Figura 11: Distribuciones de los diferentes autoencoders para la variable Temperatura del Aire

3. Metodología

Como mencionamos antes, la idea de esta tesis es proponer un método estadístico para el problema de la detección de errores en datos meteorológicos basado en autoencoders. Todas las propuestas estadísticas en esta tesis están basadas en el software estadístico R.

Como primer paso, crearemos un set de datos artificiales para explicar en que consisten los modelos de autoencoders, tener una visualización de los datos y también hacernos la idea de que es lo que está haciendo cada modelo.

3.1. Ejemplo Parábola

Como mencionamos anteriormente generamos un conjunto de 1000 datos de la siguiente manera:

$\mathbf{X} = \boldsymbol{U(0,1)}$

$\mathbf{Y} = X + \boldsymbol{N(0,0.05)}$

$\mathbf{Z} = 4(X-5)^2 + \boldsymbol{N(0,0.05)}$

Este conjunto de datos forman una parábola en $\mathbb{R}^3$, como podemos observar a continuación.

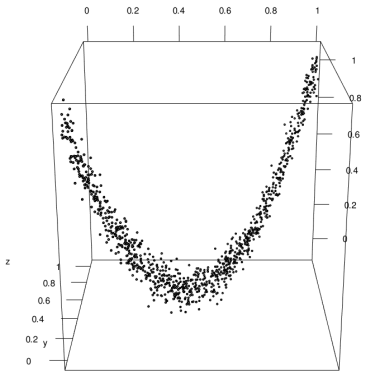


Figura 12: Conjunto de datos que simulan una parábola

Para este conjunto de datos vamos a tratar de encontrar una variedad de dimensión menor que represente de buena manera estas observaciones. Crearemos 2 modelos diferentes uno con dimensión 1 en el espacio latente y otro con dimensión 2.
Primero que nada separamos el conjunto de datos en 2 set, uno para training (70 %) y otro para test (30 %). Luego de probar varios autoencoders con diferentes profundidades en sus redes, encontramos un modelo que aproxima bien a la variedad que queríamos encontrar.

Para la selección del **score** del autoencoder vamos a usar una de las funciones más utilizadas:

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{|\mathbf{X_{ij}} - \mathbf{X'_{ij}}|}{m}$$

Y para la función de pérdida también usaremos una función muy utilizada:

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{(\mathbf{X_{ij}} - \mathbf{X'_{ij}})^2}{m}$$

Donde **X** es la matriz con los datos que creamos, **X'** es la matriz con los datos que calculamos con nuestro autoencoder, **m** es el número de columnas y **n** es el número de filas.

El modelo que construimos cuenta con las siguientes capas:

ENCODER:	**ESPACIO LATENTE:**	**DECODER:**
Capa de Entrada = 3 nodos		Quinta capa = 4 nodos
Primera capa = 6 nodos	Cuarta capa = i nodos $i \in \{1, 2\}$	Sexta capa = 5 nodos
Segunda capa = 5 nodos		Séptima capa = 6 nodos
Tercera capa = 4 nodos		Capa de Salida = 3 nodos

Además como vimos en la introducción cada capa tiene una función de activación asociada, en este caso, nosotros utilizamos la función **ReLU** es todas las capas, salvo en el espacio latente que usas la **Tanh** y en la capa de salida que usamos una función **Lineal**.

3.1.1. Espacio latente con dimensión 1

A continuación mostraremos el gráfico de los observaciones y le agregaremos los datos predichos por nuestro modelo para que podamos observar cual es la variedad que detectamos.

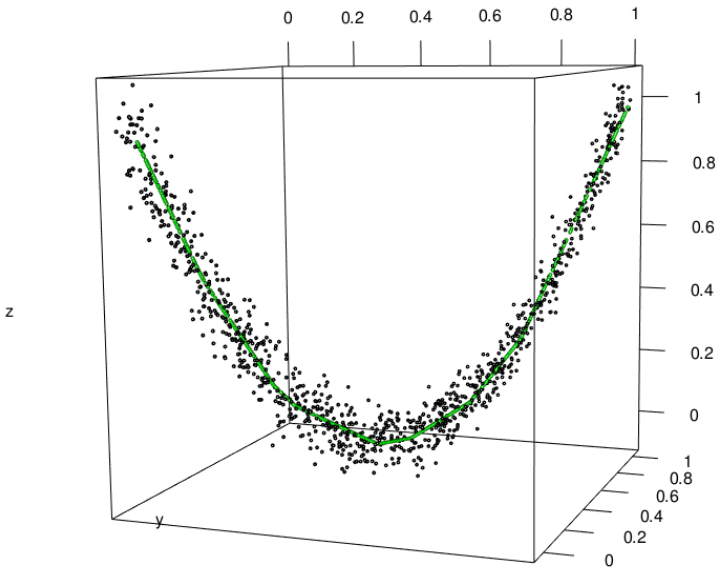


Figura 13: Conjunto de datos y los datos predichos por nuestro modelo con dimensión 1 en el espacio latente

Como podemos observar en la Figura 13 vemos que nuestro modelo encuentra efectivamente la variedad que nosotros queríamos. Además sabemos que el modelo converge ya que tanto la función de pérdida (**MSE**) como la métrica (**MAE**) que utilizamos, van disminuyendo a medida que entrenamos nuestro autoencoder. En este caso corrimos el modelo con 15 épocas, donde época indica la cantidad de veces que itera el algoritmo de optimización sobre el set de entrenamiento. En la Figura 14 podemos ver como disminuyen el MAE y el MSE.

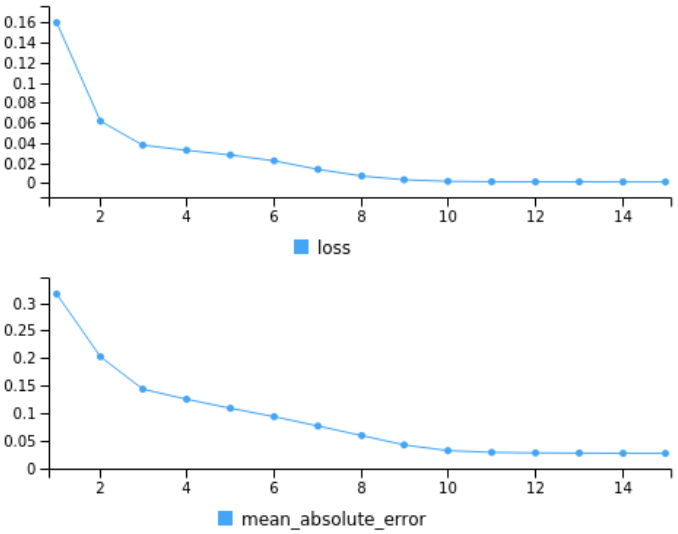


Figura 14: Valores del Score y la función de pérdida en cada época

Para ver con más claridad que hace nuestro modelo, tomamos 3 puntos del conjunto de datos (color ROJO) y los comparamos con los datos que nuestro modelo predijo (color VERDE) para dichos puntos.

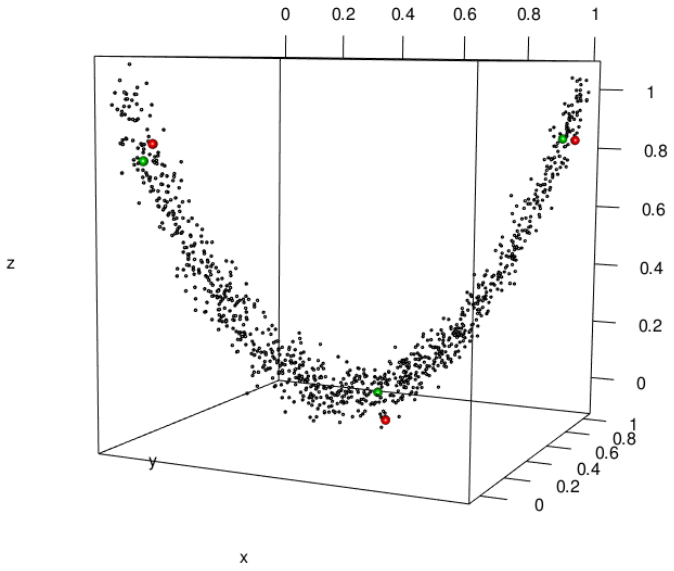


Figura 15: Muestra de 3 datos y de su predicción por el modelo con dimensión 1 en el espacio latente

En la figura anterior vemos como los puntos rojos son “empujados” hacia el centro de la parábola para formar la variedad que está en la figura 13. Esto es lo que hace nuestro modelo, agrupar los puntos de nuestro conjunto de datos inicial, en una variedad de dimensión menor.

3.1.2. Espacio latente con dimensión 2

Este mismo análisis que hicimos para el autoencoder con dimensión 1 en espacio latente, también lo hicimos para el caso de dimensión 2 y no hay grandes diferencias en los resultados. La variedad que encuentra el modelo de dimensión 1 es muy similar a la que encuentra con dimensión 2, como podemos ver en el siguiente gráfico.

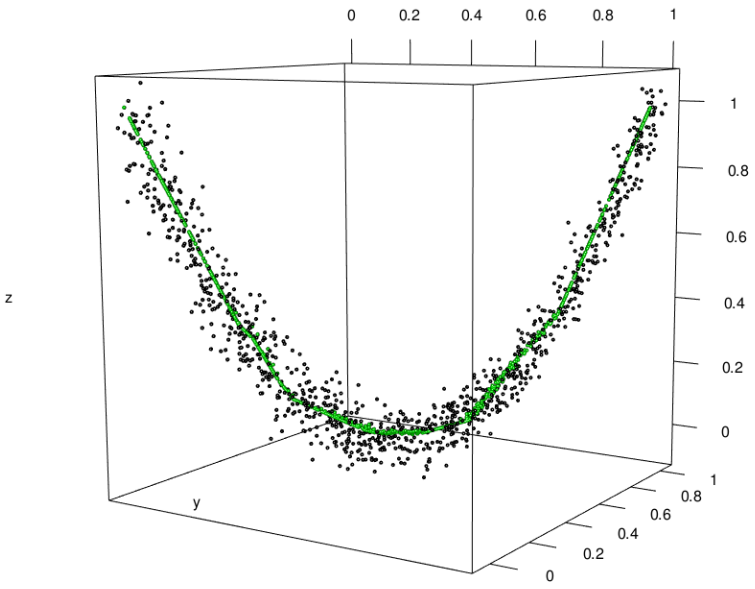


Figura 16: Conjunto de datos y los datos predichos por nuestro modelo con dimensión 2 en el espacio latente

Al igual que en el caso anterior, también sabemos que este modelo converge ya que tanto el **MSE** como **MAE** decrecen a medida que entremos el modelo.

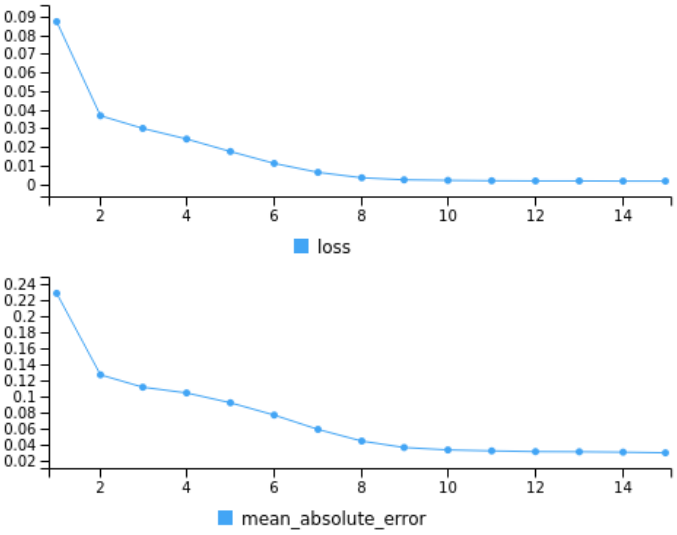


Figura 17: Valores del Score y la función de pérdida en cada época

Al haber encontrado una variedad similar, nuestro autoencoder con dimensión 2 en el espacio latente, también está "empujando" los puntos hacia el centro de la parábola. Igualmente representamos los 3 puntos del conjunto y sus 3 datos predichos para corroborarlo y efectivamente es así, como podemos observar en el siguiente gráfico.

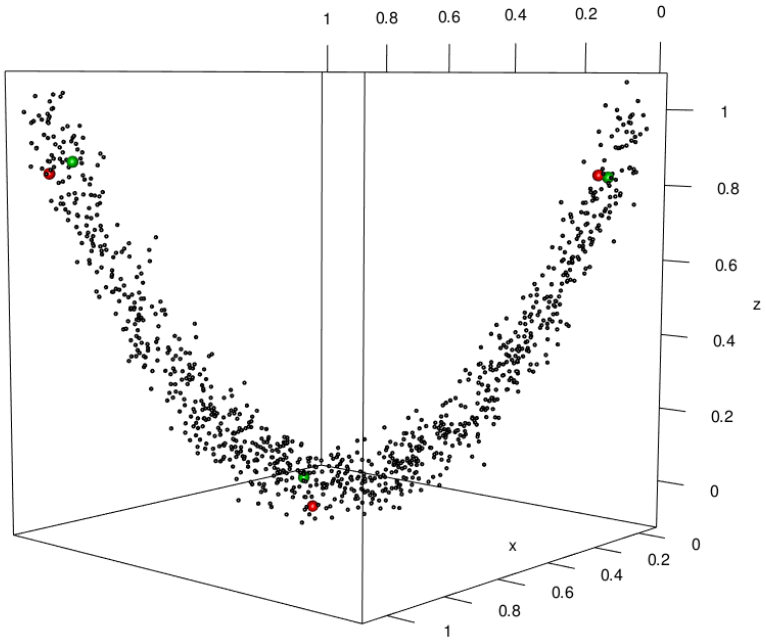


Figura 18: Muestra de 3 datos y de su predicción por el modelo con dimensión 2 en el espacio latente

Como podemos ver, en este caso no hay una gran diferencia entre los modelos con espacio latente de dimensión 1 y 2, ya que la variedad que queríamos encontrar estaba en dimensión 1. Pero, ¿qué ocurría si la dimensión del espacio latente de nuestro modelo es más chica que la dimensión de la variedad donde se encuentran los datos? ¿El modelo aproximará bien? Para contestar estás preguntas creamos un nuevo ejemplo que analizaremos a continuación.

3.2. Ejemplo Paraboloide

En este ejemplo, generamos un conjunto de 10000 datos de la siguiente manera:

X = ***U(0,1)***

Y = ***U(0,1)***

Z = $4(X-5)^2 + 4(Y-5)^2$ + ***N(0,0.05)***

Este conjunto de datos forman un paraboloide en $\mathbb{R}^3$, como podemos observar a continuación.

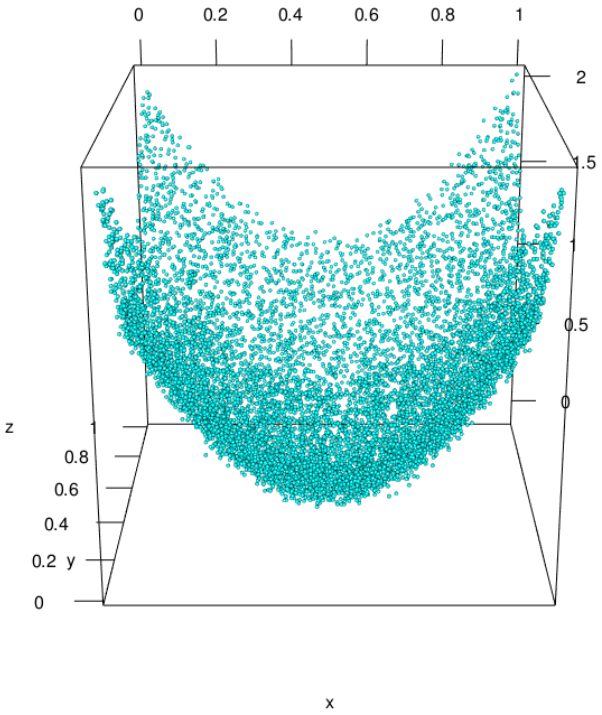


Figura 19: Conjunto de datos que simulan un paraboloide

Para este conjunto de datos vamos a tratar de encontrar una variedad que represente de buena manera las observaciones. Crearemos 3 modelos con diferentes dimensiones en el espacio latente, con dimensión 1, 2 y 3. Primero que nada separamos el conjunto de datos en 2 set, uno para training (70 %) y otro para test (30 %).

Lo que queremos mostrar con este ejemplo, es que como el conjunto de datos vive en una superficie de dimensión 2, el modelo que buscamos, con dimensión 1 en el espacio latente, va a tener un error de reconstrucción grande ya que no vamos a poder aproximar con una curva a una superficie de dimensión 2.

Basándonos en trabajos previos y luego de probar diversos autoencoders con diferentes profundidades, nodos y funciones de activación sus redes, encontramos un modelo que aproxima bien a la variedad que queríamos encontrar que es el objetivo de nuestra tesis. Cabe destacar que en este caso elegimos la $Tanh$ como función de activación en la capa que pasa al espacio latente debido a que nuestros datos tenían forma curva y la función $ReLU$ no nos ofrecía un buen rendimiento. El modelo que construimos cuenta con las siguientes capas:

ENCODER:	**ESPACIO LATENTE:**	**DECODER:**
Capa de Entrada = 3 nodos		Sexta capa = 5 nodos
Primera capa = 40 nodos		Séptima capa = 10 nodos
Segunda capa = 20 nodos	Quinta capa = i nodos $i \in \{1, 2, 3\}$	Octaba capa = 20 nodos
Tercera capa = 10 nodos		Novena capa = 40 nodos
Cuarta capa = 5 nodos		Capa de Salida = 3 nodos

Usamos la función de activación **ReLU** en todas las capas, salvo en el espacio latente que usamos la **Tanh** y en la capa de salida que usamos una función **Lineal**.

3.2.1. Espacio latente con dimensión 2

A continuación mostraremos el gráfico de los observaciones y le agregaremos los datos predichos(color ROJO) por nuestro modelo para que podamos observar que la variedad que encontramos aproxima bien a los datos.

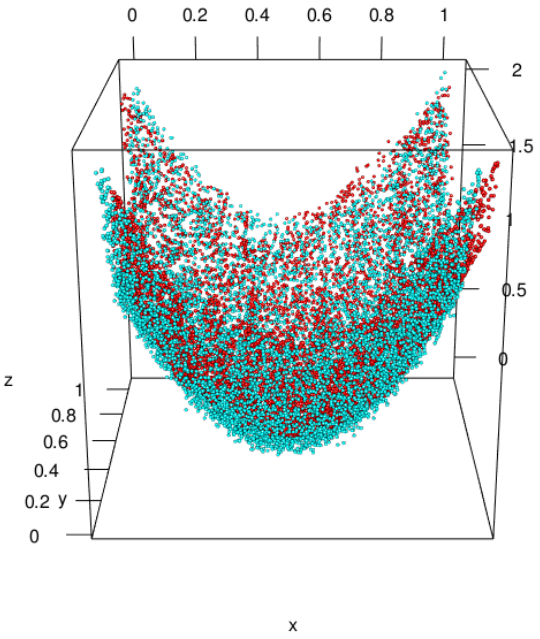


Figura 20: Conjunto de datos y sus datos predichos por nuestro modelo con dimensión 2 en el espacio latente

Como podemos observar en la figura 20 vemos que nuestro modelo encuentra efectivamente la variedad que nosotros queríamos. Pero al tener tantos puntos no se ve claramente, por lo que graficaremos la superficie que genera encuentra nuestro modelo y le superpondremos 1000 datos con un ruido mayor ***N(0,0.15)***.

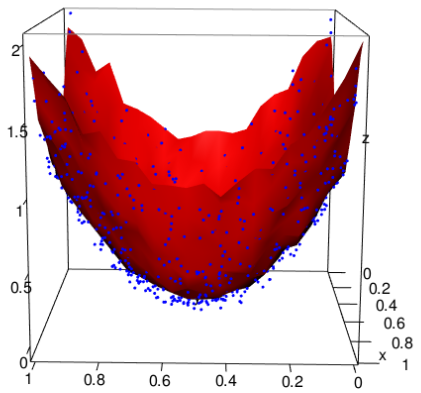


Figura 21: Conjunto de datos y nuestro modelo predicho con dimensión 2 en el espacio latente

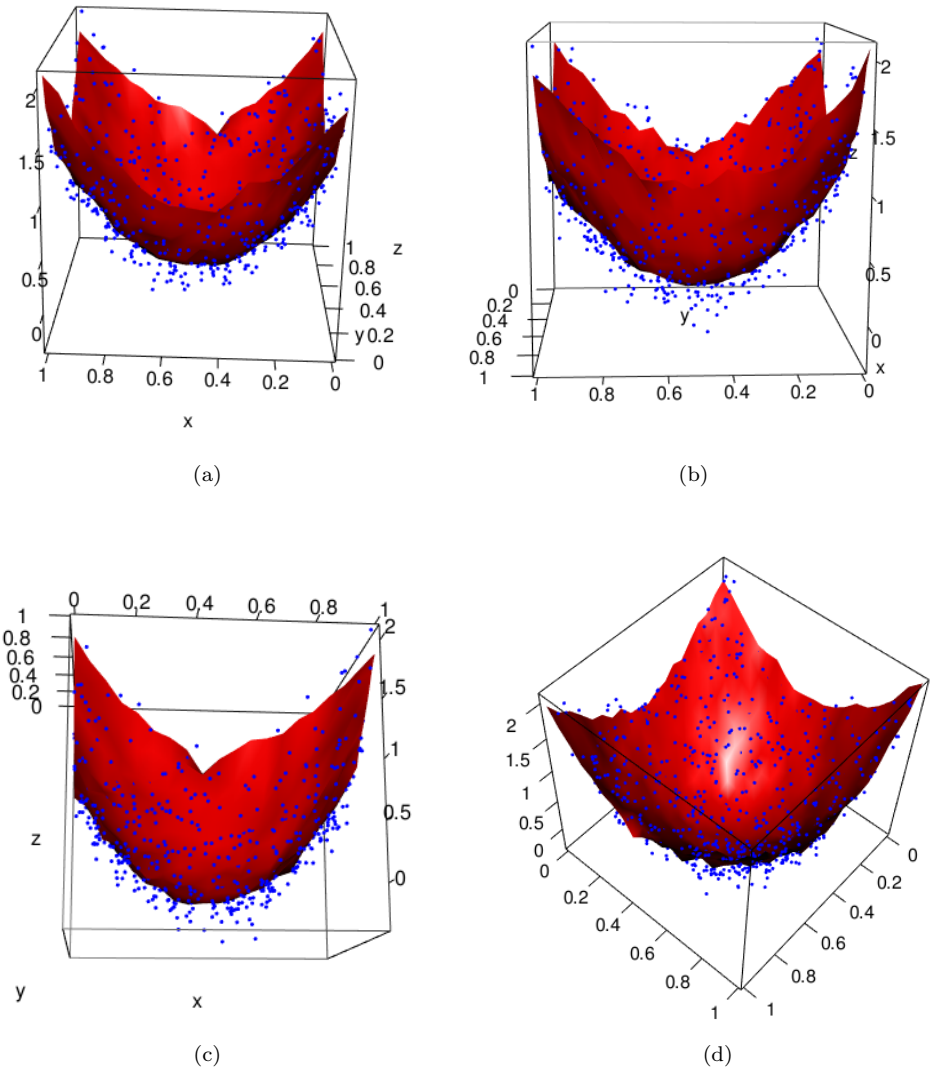


Figura 22: Superficie de los datos predichos por el modelo y 1000 datos con más ruido en diferentes angulos

Además sabemos que el modelo converge ya que tanto la función de pérdida (**MSE**) como la métrica (**MAE**) que utilizamos, van disminuyendo a medida que entrenamos nuestro autoencoder, en este caso corrimos el modelo con 15 épocas como podemos visualizar en la siguiente imagen.

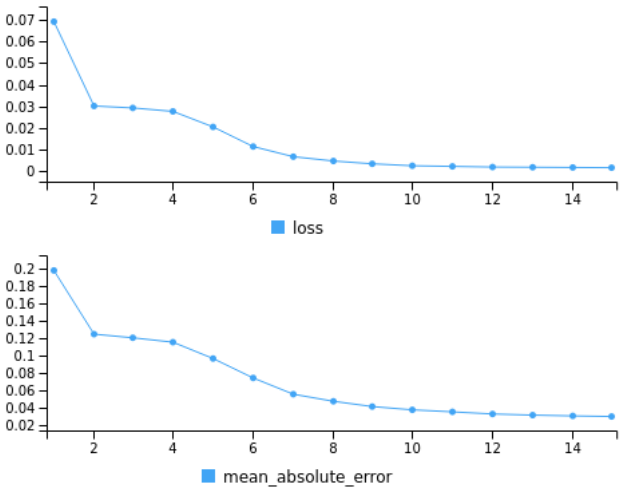


Figura 23: Valores del Score y la función de pérdida en cada época

En el siguiente gráfico, tomamos 5 puntos del conjunto de datos(color VERDE) y los comparamos con los datos que nuestro modelo predijo(color ROJO) y como podremos observar, la aproximación es muy buena. Para poder ver los puntos seleccionados graficaremos la imagen vista desde otra perspectiva.

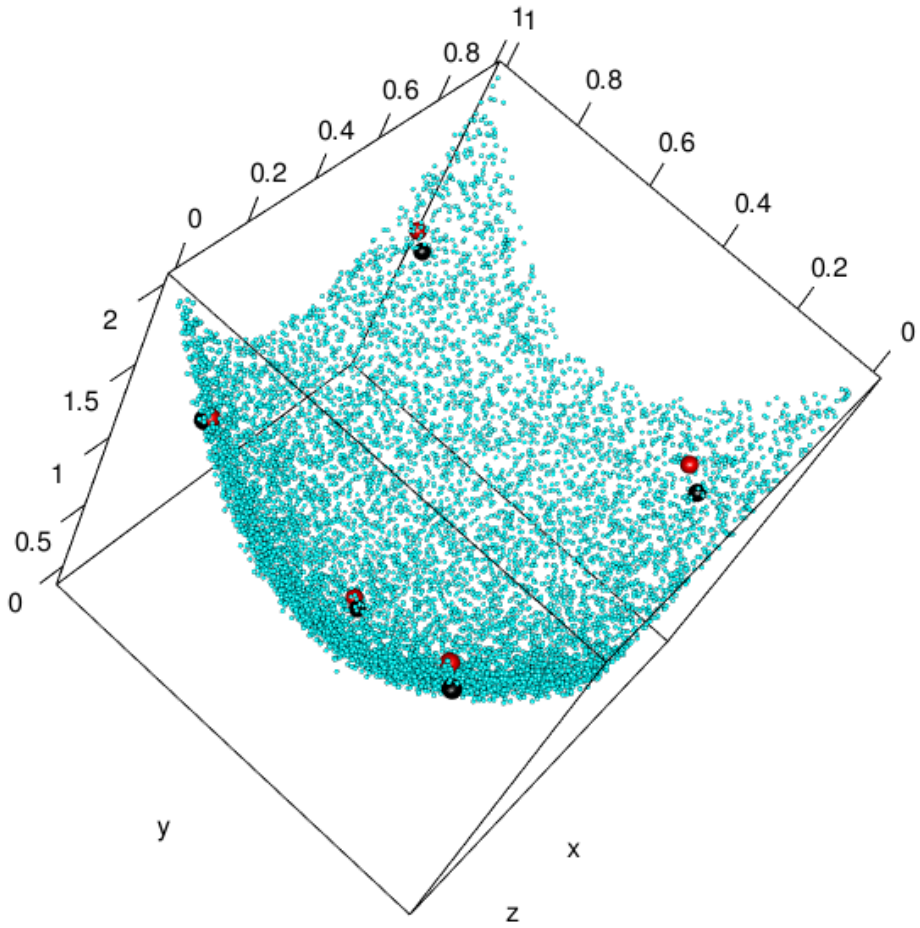


Figura 24: Muestra de 5 datos y de su predicción por el modelo con dimensión 2 en el espacio latente

3.2.2. Espacio latente con dimensión 3

Este mismo análisis que hicimos para el autoencoder con dimensión 2 en espacio latente, también lo hicimos para el caso de dimensión 3 y los resultados fueron perecidos. La variedad que encuentra el modelo de dimensión 2 es muy similar a la que encuentra con dimensión 3.

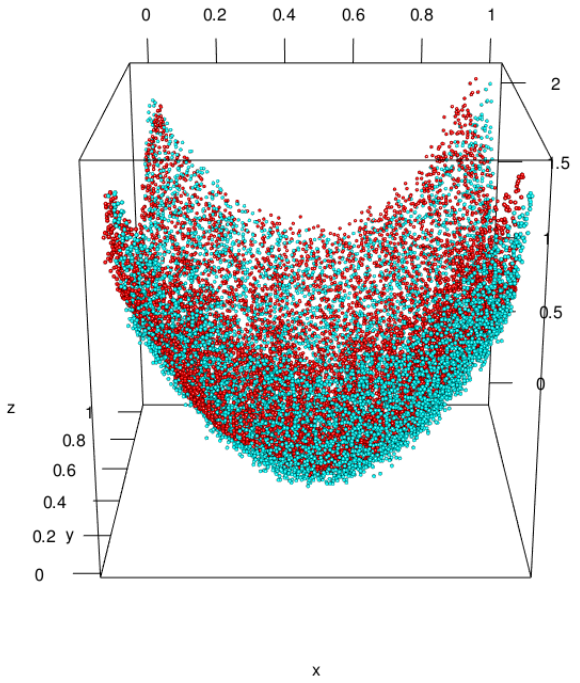


Figura 25: Conjunto de datos y sus datos predichos por nuestro modelo con dimensión 3 en el espacio latente

Para este caso tomaremos los mismos 1000 datos que usamos antes, graficaremos la superficie que encuentra nuestro modelo y también le superpondremos estos datos con el ruido aumentado como en el caso anterior.

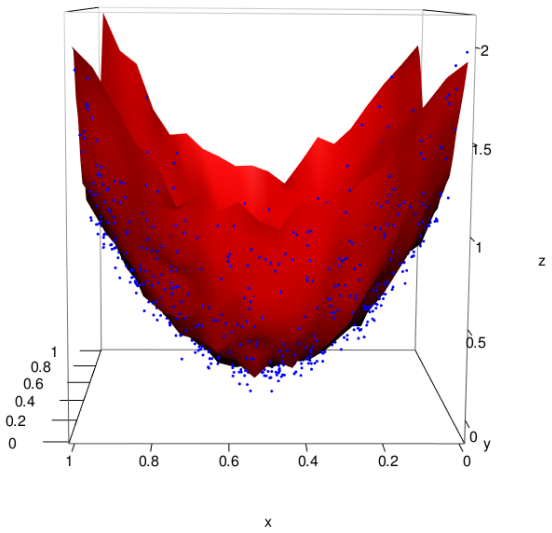


Figura 26: Superficie de los datos predichos por el modelo y 1000 datos con ruido

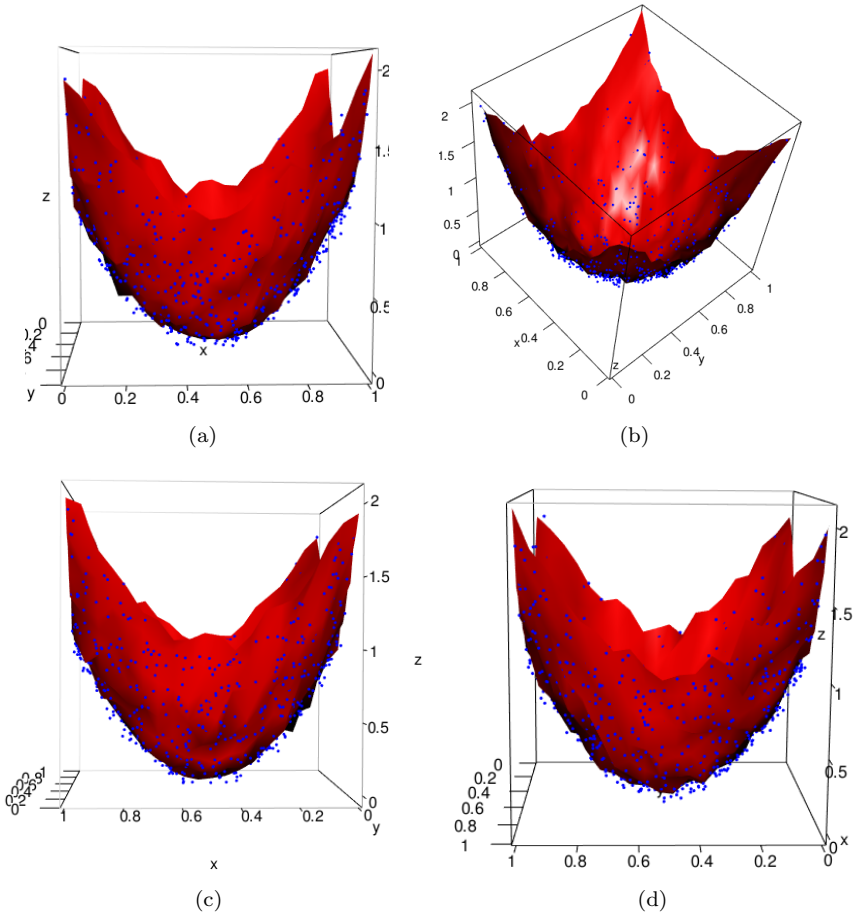


Figura 27: Superficie de los datos predichos por el modelo y 1000 datos con más ruido en diferentes ángulos

Al igual que en el caso anterior, también sabemos que este modelo converge ya que tanto el **MSE** como **MAE** decrecen a medida que entremos el modelo.

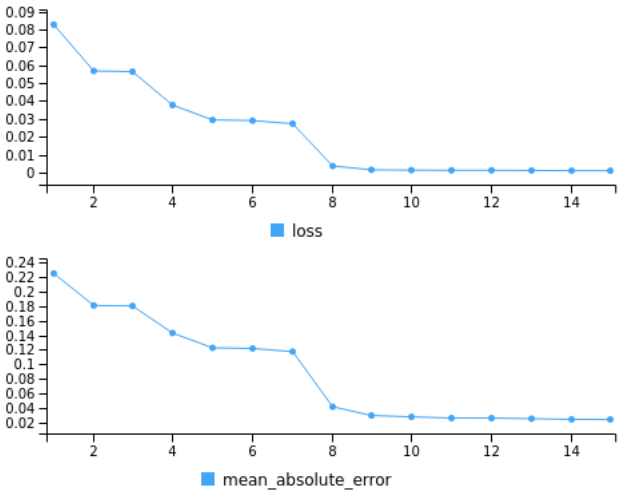


Figura 28: Valores del Score y la función de pérdida en cada época

Al haber encontrado una variedad similar, nuestro autoencoder con dimensión 3 en el espacio latente, también está aproximando bien la superficie del paraboloide. Igualmente representamos los 5 puntos

del conjunto(NEGRO) y sus 5 datos predichos(ROJO) para corroborarlo y efectivamente es así, como podemos observar en el siguiente gráfico.

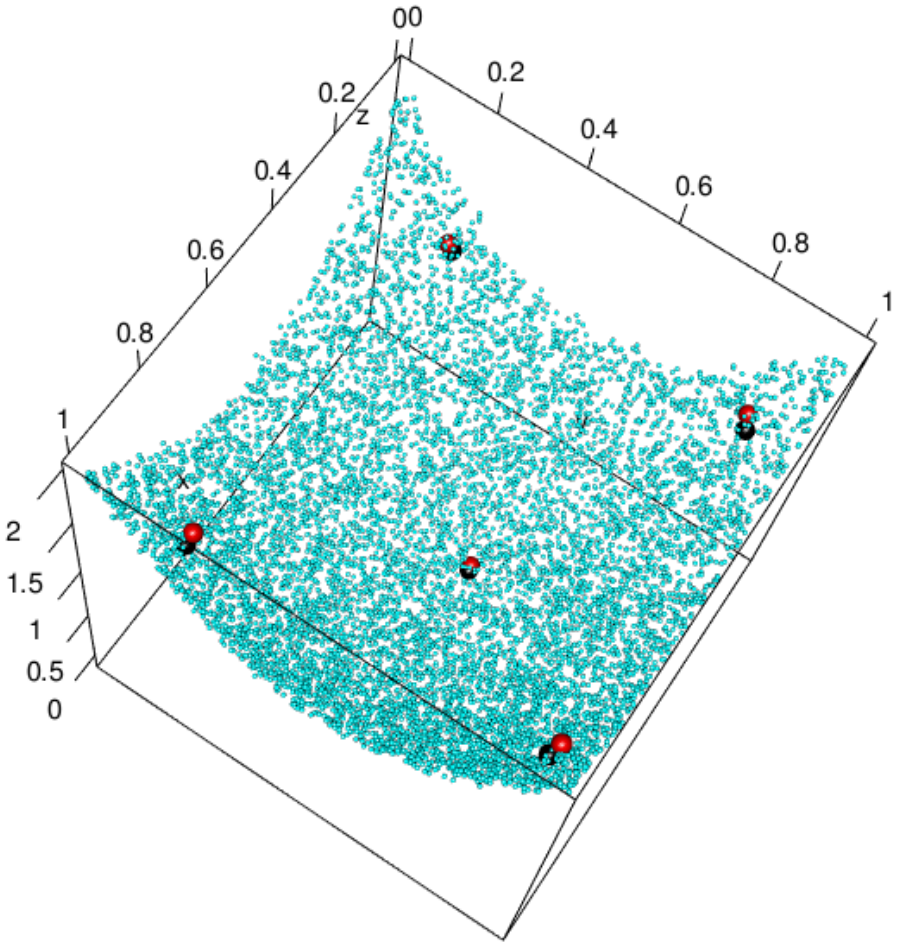
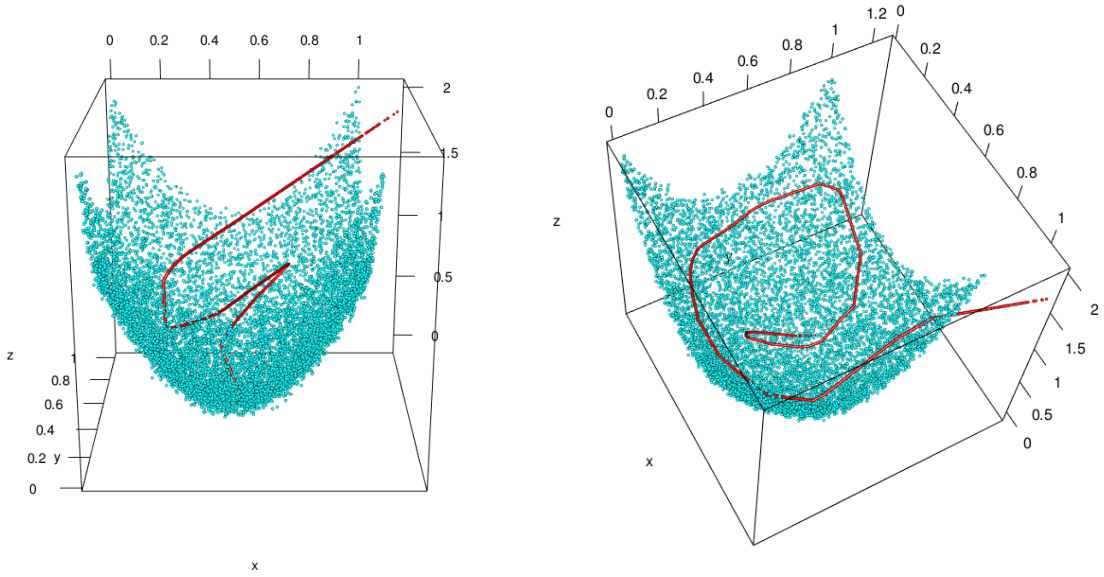


Figura 29: Muestra de 5 datos y de su predicción por el modelo con dimensión 3 en el espacio latente

3.2.3. Espacio latente con dimensión 1

Al igual que con dimensión 2 y 3 en el espacio latente también analizamos el caso con dimensión 1, pero como era de esperarse los resultados no son muy buenos, ya que no vamos a lograr una buena aproximación con un curva a una superficie de dimensión 2. A continuación mostraremos 2 entrenamientos diferentes de nuestro modelo.



(a) (b)

Figura 30: Conjunto de datos y sus datos predichos por 2 de nuestros modelos con dimensión 1 en el espacio latente

En estos caso ni el **MSE** ni **MAE** disminuyen de manera significativa debido a que el modelo no aproxima bien a los datos, esto se ve claramente en el siguiente gráfico, el error de reconstrucción no llega a ser ni la mitad de lo que era en el punto inicial.

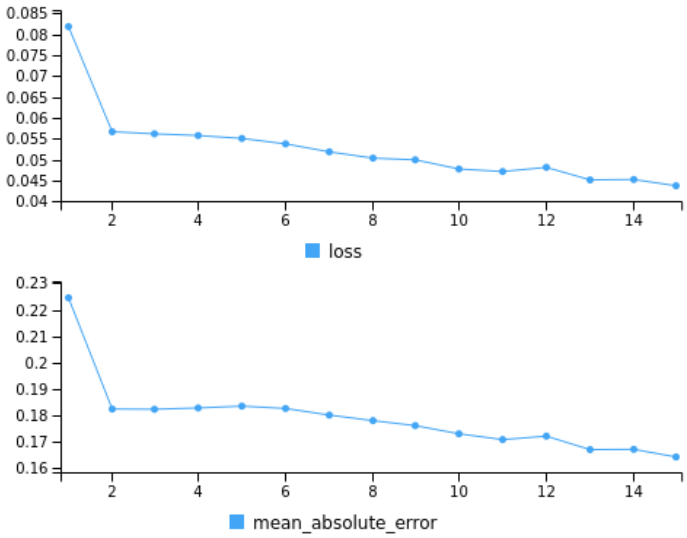


Figura 31: Valores del Score y la función de pérdida en cada época

Como vimos anteriormente, la aproximación de este modelo no es muy buena, por lo que en este caso, graficaremos los datos predichos por nuestro modelo y también la proyección de 5 puntos para ver a que lugar de la variedad van a parar.

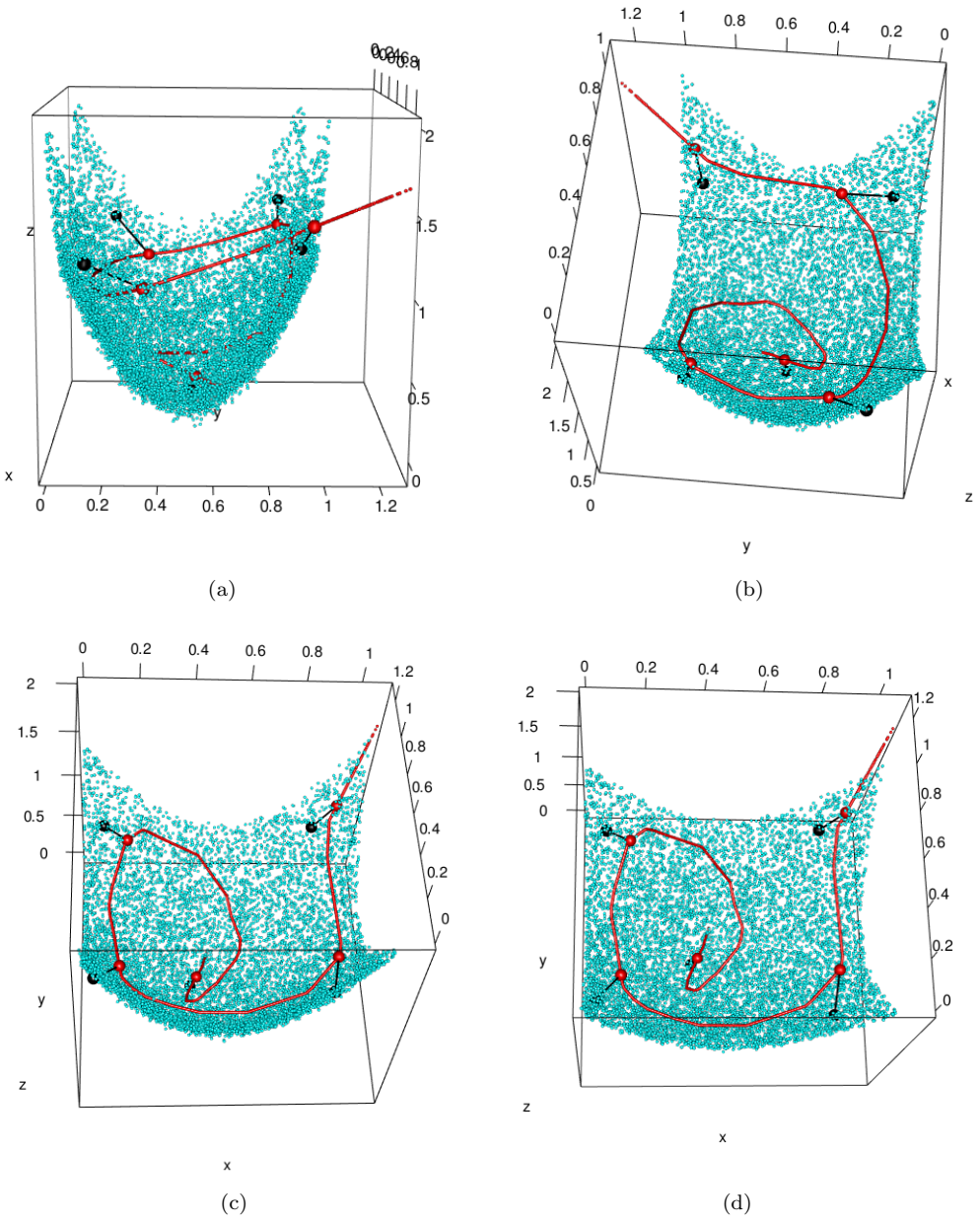


Figura 32: Muestra de nuestro modelo y 5 datos más su predicción

Aún cuando vemos que la aproximación de los datos no es la mejor, podemos observar que el modelo recorre la superficie, lo que nos dice que el entrenamiento funciona, el problema está en que la dimensión del espacio latente es demasiado pequeña para poder representar los datos de mejor manera.

3.3. Elección del espacio latente

Para encontrar cual es el mejor espacio latente, vamos a utilizar el error de reconstrucción (**mae**). Lo que haremos es para cada dimensión del espacio latente(1, 2 y 3) correremos 100 veces nuestro modelo y en cada corrida iremos guardando por columna en una matriz, los errores de reconstrucción del set de datos test, es decir, en la primer columna tendríamos todos los errores de reconstrucción de la corrida número 1, por lo que tendríamos 3000 filas(los datos test son el 30 % de los 10000 datos). Al final obtendremos una matriz que vive en $\mathbb{R}^{3000\times 100}$ para cada espacio latente, haciendo un total de 3 matrices.

Una vez que tenemos construidas nuestras matrices, lo que hacemos es tomar la mediana por columnas, así obtenemos la mediana de los errores de reconstrucción para cada corrida del modelo. Es muy importante que tomemos la mediana y no la media, porque ésta es más robusta y no se verá tan afectada si en algún caso el error de reconstrucción es alto, lo cual es muy posible, ya que como la inicialización de los pesos es estocástica, el modelo en algunos casos podría no converger. Por lo tanto ahora tendremos 3 vectores de longitud 100, con las distintas medianas de los **mae**, a continuación graficaremos las distribuciones de estos distintos vectores.

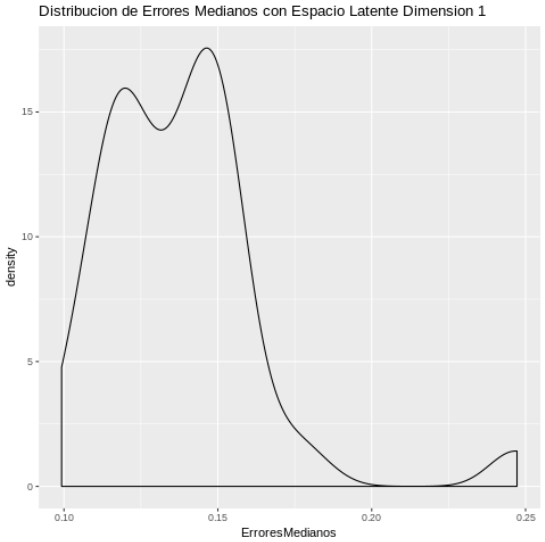


Figura 33: Función de densidad de los errores medianos del modelo con espacio latente de dimensión 1

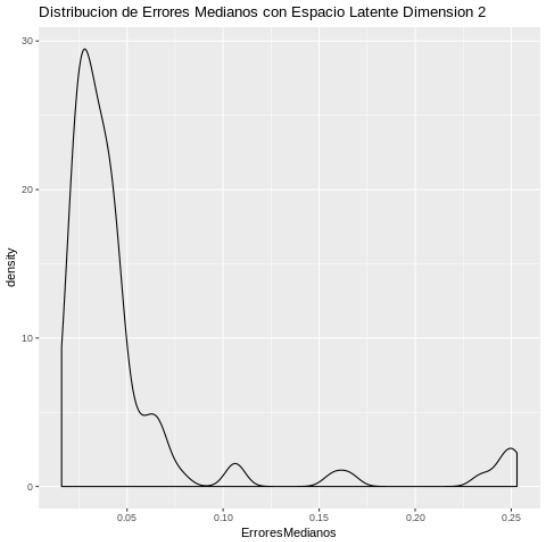


Figura 34: Función de densidad de los errores medianos del modelo con espacio latente de dimensión 2

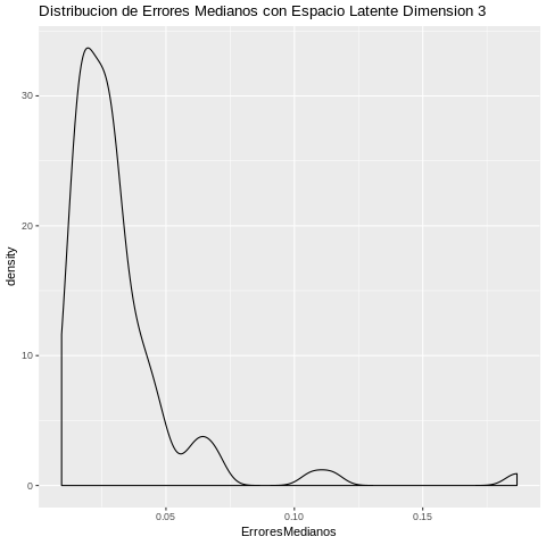


Figura 35: Función de densidad de los errores medianos del modelo con espacio latente de dimensión 3

Como podemos observar, las distribuciones para los modelos con dimensión 2 y 3 son similares y en la mayoría de los casos el error de reconstrucción es pequeño, pero para el caso de dimensión 1 vemos que el error de reconstrucción es grande. Por último tomaremos la mediana de cada unos de los 3 vectores para las diferentes dimensiones del espacio latente, a este error lo llamaremos **Error Mediano Absoluto** y es el que usaremos para identificar cual es el mejor espacio latente.

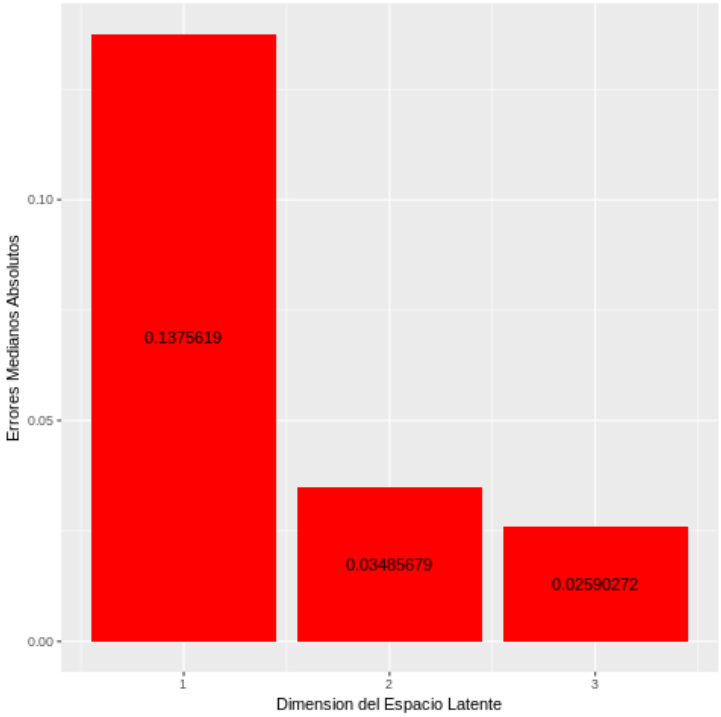


Figura 36: Errores Medianos Absolutos de cada uno de nuestros modelos

Como observamos en el gráfico anterior, hay un salto muy grande en el **Error Mediano Absoluto** entre el espacio latente de dimensión 1 y el de dimensión 2 y la diferencia entre los que tienen dimensión 2 y 3 es bastante poca.

En este ejemplo, la conclusión de que espacio latente elegir es el de dimensión 2 y es bastante claro porque elegimos este, ya que el error se reduce drásticamente de dimensión 1 a dimensión 2 y si luego si al de dimensión 2 le agregamos otra neurona, no cambia mucho. En este caso no fue difícil identificar el espacio latente, pero cuando tengamos más dimensiones puede que la elección del "mejor" espacio latente no sea tan simple.

4. Aplicación a Datos Reales

Una vez hecho el análisis anterior para los conjuntos de datos que fabricamos, vamos a usar todo lo aprendido en un conjunto de datos reales. La base del método la vamos a construir en función de un conjunto de datos de series de tiempo registrado en la Estación Meteorológica del Instituto Max Planck de Biogeoquímica en Jena, Alemania.

4.1. Análisis descriptivo datos

En este set de datos, se registraron 14 variables (como temperatura del aire, presión atmosférica, humedad, dirección del viento, etc) cada 15 minutos, durante varios años. Los datos originales, van desde 2003, pero en este caso de estudio nos limitamos a los datos entre los años 2009-2016. Este conjunto de datos es perfecto para aprender a trabajar con series de tiempo numéricas.

La variable donde teníamos la fecha la transformamos en 4 variables nuevas que a continuación explicaremos en detalle. Las 14 variables originales son las siguientes:

Var1) **p**: Presión de Aire en milibar (mbar)

Var2) **T**: Temperatura del Aire en grados Celcius

Var3) **Tpot**: Temperatura Potencial en grados Kelvin

Var4) **Tdew**: Temperatura de Rocio

Var5) **rh**: Porcentaje de Humedad Relativa(%).

Var6) **VPmax**: Presión de vapor de saturación (mbar), es la presión a la cual el fluido pasa del estado gaseoso al líquido(o del líquido al gaseoso) para una temperatura dada

Var7) **VPact**: Presión de Vapor (mbar)

Var8) **VPdef**: Déficit de Presión del Vapor (mbar) es la diferencia entre la cantidad de humedad en el aire y cuanta humedad puede contener el aire cuando está saturado.

Var9) **sh**: Humedad Específica, es la masa de vapor de agua en una unidad de masa de aire húmedo expresada en gramos de vapor por kilogramo de aire (g/kg).

Var10) **H20C**: Concentración de vapor de agua(mmol/mol).

Var11) **rho**: Densidad del Aire (g/m^3).

Var12) **wv**: Velocidad del viento (m/s).

Var13) **max.wv**: Máxima velocidad el aire (m/s)

Var14) **wd**: Dirección del viento en grados.

4.2. Variables Temporales

Como mencionamos antes, vamos a usar la variable donde teníamos las mediciones en tiempo de cuando habían ocurrido las observaciones, pero no como está dada, sino que vamos a crear nuevas variables en función de ella, por lo tanto ahora vamos a tener nuevas variables que involucran la hora y el día en que fueron medidas las 14 variables anteriores. Para hacer esto, debemos tener cuidado, ya que hay que tener en cuenta, que si tenemos una medición a la hora 23:45:00, en general va a ser similar a la medición de la hora 00:00:00 del día siguiente, por la tanto el valor que le vamos a asignar a dichas mediciones en nuestra nueva variable va a tener que ser similar. Lo mismo va a ocurrir con el día 365 y el día 01 del año siguiente. La forma en la que resolvimos esto, fue hacer un cambio de variable para poder expresar mejor este fenómeno que consiste en lo siguiente:

Dada una determinada hora HH:MM:SS, la convertimos en HH,MM, es decir la hora 15:45:00, pasaría a ser el número 15,45. Una vez hecho esto, vamos escribir este número en 2 coordenadas, usando la parametrización ϕ_1 tal que $\phi_1(t) = (sin(\frac{2*\pi*t}{24}), cos(\frac{2*\pi*t}{24}))$

Al hacer esto, sabemos que tanto la primera como la segunda coordenada de la función ϕ_1 tienen período 24, de esta forma, $\phi_1(0) = \phi_1(24)$ y por ser continua, $\phi_1(0) \simeq \phi(23, 45)$.

De la misma forma lo hacemos con los días, si tenemos la fecha YYYY:MM:DD, lo que hacemos es asignarle el valor del día que le corresponde entre 0 que sería el 1 de enero y el 364 que sería el 31 de diciembre. En este caso, vamos a escribir a este número también en 2 coordenadas, usando la parametrización ϕ_2 tal que $\phi_2(t) = (sin(\frac{2*\pi*t}{365}), cos(\frac{2*\pi*t}{365}))$.

Al hacer esto, sabemos que tanto la primera como la segunda coordenada de la función ϕ_2 tienen período 24, de esta forma, $\phi_2(0) = \phi_2(365)$ y por ser continua, $\phi_2(0) \simeq \phi_2(364)$.

Por lo tanto, vamos a tener 4 nuevas variables, siendo ahora un total de 18 y con este conjunto de datos es con el que realizaremos el análisis.

4.2.1. Control de consistencia

El método de detección de errores propuesto, cuenta con una parte previa que se conoce como consistencia, donde hacemos un control de consistencia univariado de los datos. Esto se debe a que a que las redes neuronales no soportan datos de magnitudes muy grandes. El control lo hacemos de la siguiente forma, calculamos la **mediana** y el **MAD** de cada variable y las estandarizamos, luego eliminamos aquellos datos en los cuales tengamos una variable cuyo valor absoluto sea mayor que 7; esto lo hacemos para eliminar outliers extremos univariados ya que como el escalamiento que hicimos es robusto, lo normal es que los datos en cada variable estén entre -3 y 3 aproximadamente.

Donde el **MAD** es:

Dado $X = (X_1, ..., X_n)$ con $X_1, ..., X_n$ univariados y $\tilde{X} = median(X)$

$MAD = median(|X_i - \tilde{X}|)$

4.2.2. Marco principal

A continuación describiremos como es el marco principal que seguiremos en el desarrollo de la metodología para la detección de errores en datos meteorológicos.

- Separamos nuestros datos en 2 conjuntos: train, test. Los datos de train los vamos a usar para entrenar nuestro modelo y los datos de test los usaremos para ver que tan bueno es.
- Construiremos un primer modelo de autoencoder para detectar un error en forma general.
- Construiremos 14 nuevos autoencoders, cada uno quitando una variable distinta.
- Compararemos el error calculado con el método general con los errores calculados por los 14 autoencoders, para detectar cual es la variable que introduce el error.

Para la selección del **score** del autoencoder vamos a usar:

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{|\mathbf{X_{ij}} - \mathbf{X'_{ij}}|}{m}$$

Pero también vamos a usar:

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{(\mathbf{X_{ij}} - \mathbf{X'_{ij}})^2}{m}$$

$$\mathbf{mae}_i = \frac{1}{m}\sum_{j=1}^{m}|\mathbf{X_{ij}} - \mathbf{X'_{ij}}|$$

Donde **X** es la matriz con los datos originales y **X'** es la matriz con los datos que calculamos con nuestro autoencoder, **m** es el número de columnas y **n** es el número de filas.

4.3. Modelo

Nuestro modelo constará de 1 capa de entrada, 7 capas intermedias, donde en cada una de ellas la función de activación será la función **ReLU** y al final una capa de salida donde usaremos una función lineal. A continuación mostramos como sería el modelo y cuantas nodos usamos en cada capa.

ENCODER:	**ESPACIO LATENTE:**	**DECODER:**
Capa de Entrada = 18 nodos		Quinta capa = 32 nodos
Primera capa = 128 nodos	Cuarta capa = i nodos $i \in \{3, ..., 18\}$	Sexta capa = 64 nodos
Segunda capa = 64 nodos		Séptima capa = 128 nodos
Tercera capa = 32 nodos		Capa de Salida = 18 nodos

4.4. Espacio latente

Para elegir el mejor espacio latente, vamos a utilizar la metodología que vimos anteriormente, la cual consistía en correr 100 veces nuestro modelo para cada dimensión del espacio latente y para cada corrida guardábamos los errores de reconstrucción de nuestro conjunto de datos test, por lo que ahora tendríamos que cada matriz vivirá en $\mathbb{R}^{123611\times 100}$. Luego a cada una de estas matrices calculamos el **Error Mediano Absoluto**, que sería tomar las medianas de cada matriz por columna y luego a este vector calcularle su mediana. En este caso tomaremos las dimensiones del espacio latente desde la dimensión 3 hasta la dimensión del espacio total, las dimensiones 1 y 2 no las consideramos ya que para el caso de dimensión 3 el error ya nos da muy grande.

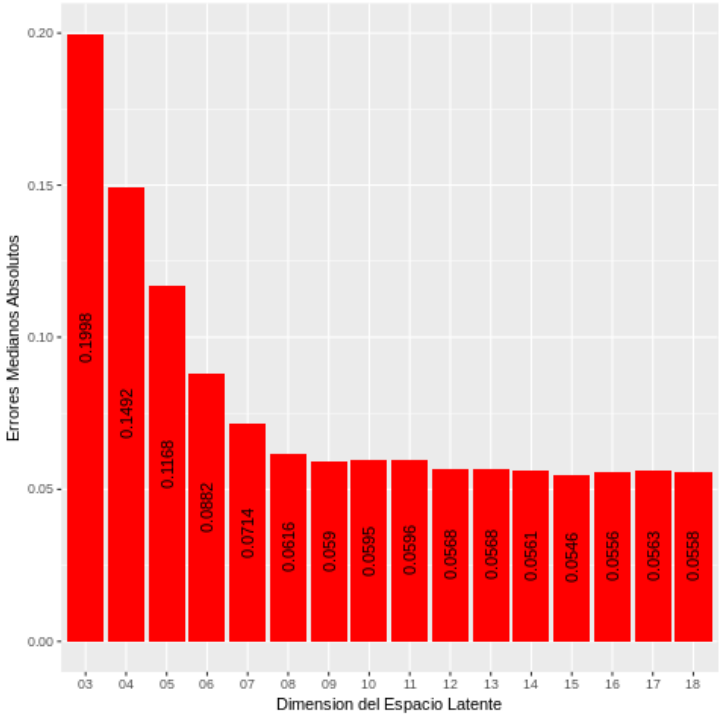


Figura 37: Errores Medianos Absolutos de cada uno de nuestros modelos

Como se puede observar en el gráfico anterior, vemos que a medida que aumentamos la dimensión de nuestro espacio latente el **Error Mediano Absoluto** va disminuyendo hasta llegar a dimensión 8 donde parece estancarse y ya no se ve un cambio significativo, por lo que en este caso consideramos que la mejor dimensión para nuestro espacio latente es dimensión 8.

4.5. Análisis de datos correctos vs erróneos

Una vez que ya tenemos elegida la dimensión del espacio latente, tenemos el "mejor" autoencoder con todas las variables. Como consideramos que los datos provenientes de Jena, Germany son "correctos", lo que hicimos fue alterar al azar algunos datos del conjunto test para ver si podíamos identificar estos errores. Estas alteraciones las hacemos univariadas, pues suponemos que los errores provienen de una sola variable y no de una combinación entre ellas.

Tomamos al azar 2000 datos del conjunto test y los modificamos de la siguiente forma. Primero calculamos el desvío de la i-ésima variable y luego modificamos cada lugar por un valor que estuviera dentro de su rango, pero con la condición de que la diferencia entre el valor absoluto del dato nuevo y el viejo sea mayor o igual a el desvío que calculamos antes. De esta forma, nos aseguramos de que el valor que estamos modificando sea un valor distinto, pero que a su vez esté en el rango de la variable, ya que si fuera un valor muy atípico, al modelo le sería muy fácil reconocerlo.

4.5.1. MSE y MAE

Una vez hecho esto, lo que tenemos en cada paso, son 2 matrices de $\mathbb{R}^{2000\times 14}$ de las cuales una tiene todos los datos normales y la otra tiene alterada la i-ésima columna(la variable i), luego calculamos el **MAE** y el **MSE** con el autoencoder que construimos para cada una de ellas. Por lo tanto, en cada paso vamos a tener 2 **MAE**, uno correspondiente a los datos normales y otro a los datos alterados, lo mismo con **MSE**. Finalmente, como haremos esto para cada variables, tendremos 28 **MAE** y 28 **MSE**, donde 14 de cada uno de ellos corresponde a los datos correctos y los otros 14 a los datos alterados. En los siguientes gráficos podemos observar las diferencias, donde el eje **x** nos dice cual fue la variable que alteramos, el eje **y** cual es su **MAE** o su **MSE** y el color nos dice si el valor hallado corresponde a los datos correctos o a los datos alterados.

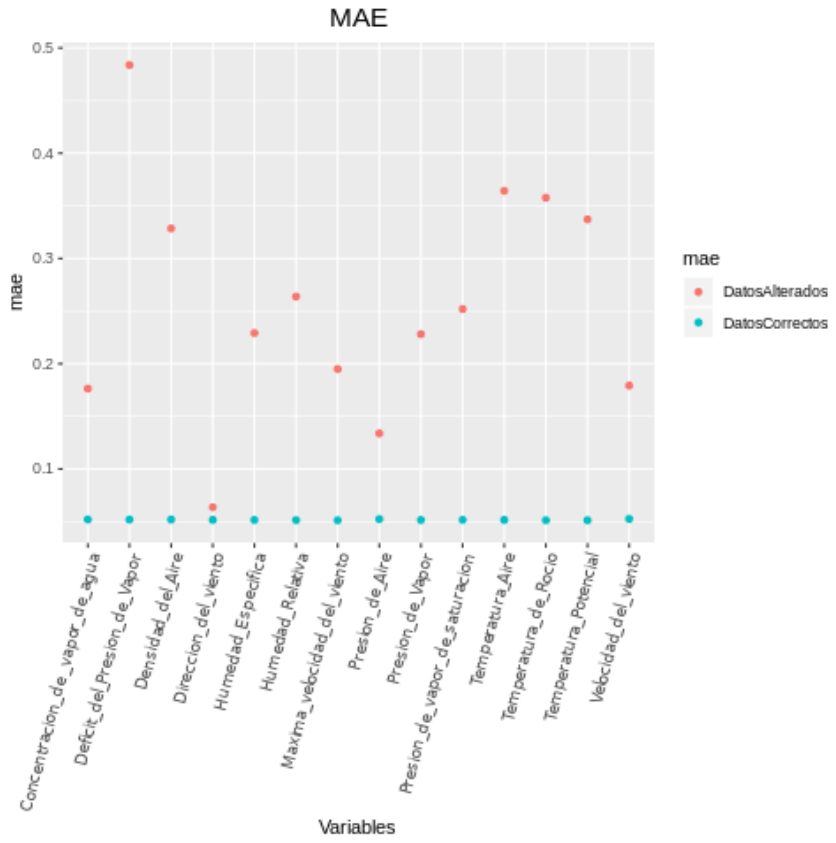


Figura 38: MAE de los datos correctos y alterados

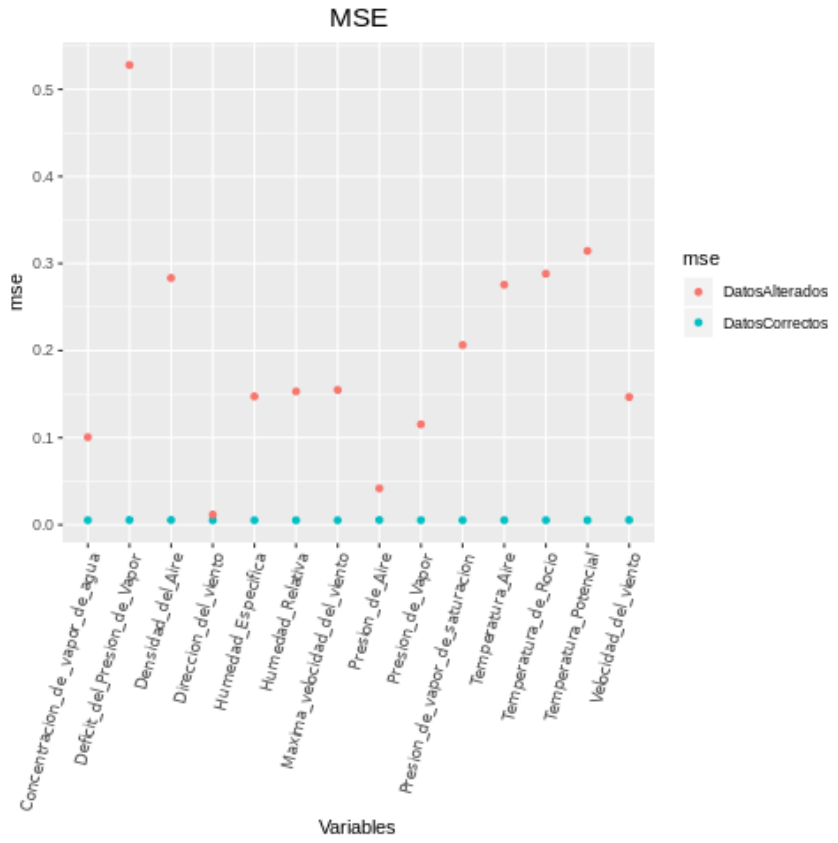


Figura 39: MSE de los datos correctos y alterados

4.5.2. Método para identificar la variable que introduce el error

El análisis que realizamos anteriormente nos sirve para identificar un error en forma general, pero también nos interesa saber cual es la variable que introduce este error, para poder identificarla lo que hacemos es construir los 14 autoencoders, donde el autoencoder i, no tiene la i-ésima variable y después definimos una función que hace lo que mencionaremos a continuación.

Calculamos el **mae** de cada dato alterado con el autoencoder con todas las variables y con cada uno de los 14 autoencoders. Hacemos la diferencia entre cada uno de estos valores. Si esta diferencia es positiva en el caso i, significa que el autoencoder sin la variable i tiene un **mae** menor, es decir es mejor. En este caso si llamamos **d** a esta diferencia, guardamos el valor $\frac{\mathbf{d}}{\mathbf{d+1}}$. En el caso de ser negativa me dice que el autoencoder funciona peor, lo cual no nos es de importancia. Hacemos esta diferencia para cada autoencoder y nos quedamos con el máximo de estos valores, que me dice cual es el i de donde proviene el error, pues es el lugar en donde difieren más las distancias (el lugar donde ésta se achicó más). Este procedimiento se lo aplicamos a cada matriz de los datos alterados, es decir, en el i-ésimo paso, tendremos una matriz de $\mathbb{R}^{2000\times 14}$ con la columna i alterada y lo que haremos es llevar acabo el procedimiento que describimos para cada fila de la matriz, al final de i-ésimo paso, lo que tendremos es un vector de longitud 2000 con números del 1 al 14 que me dirá que si en el lugar j está el número k, significa que el error correspondiente a la j-ésima observación proviene de la variable k. El siguiente gráfico, es un gráfico de barras en el cual mostramos la tasa de detección de nuestro método para encontrar la variable errónea, donde el eje **x** nos dice cual es la variable errónea de la matriz con la cual trabajamos y en el eje **y** cual es el la tasa de detección de nuestro método.

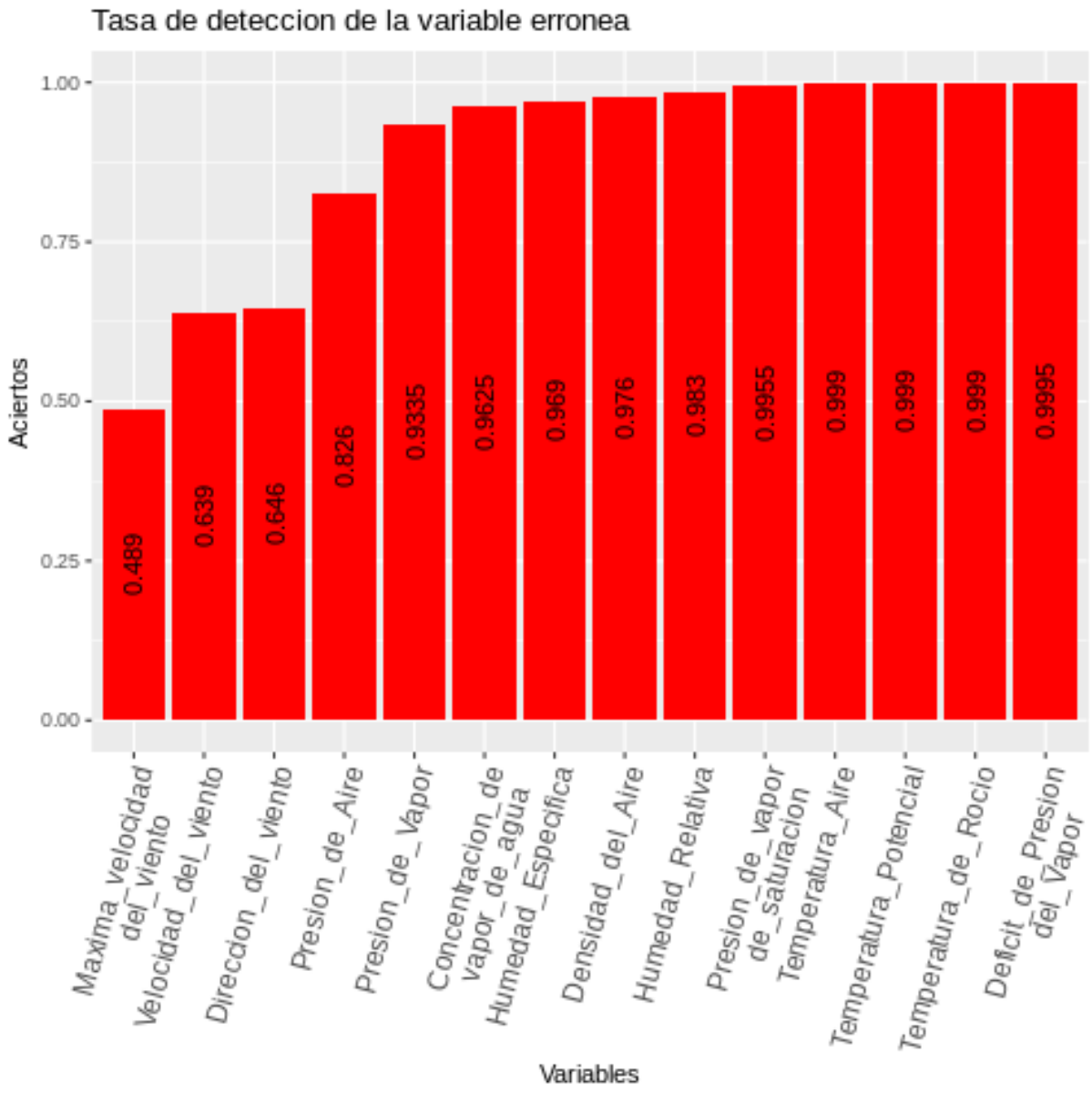
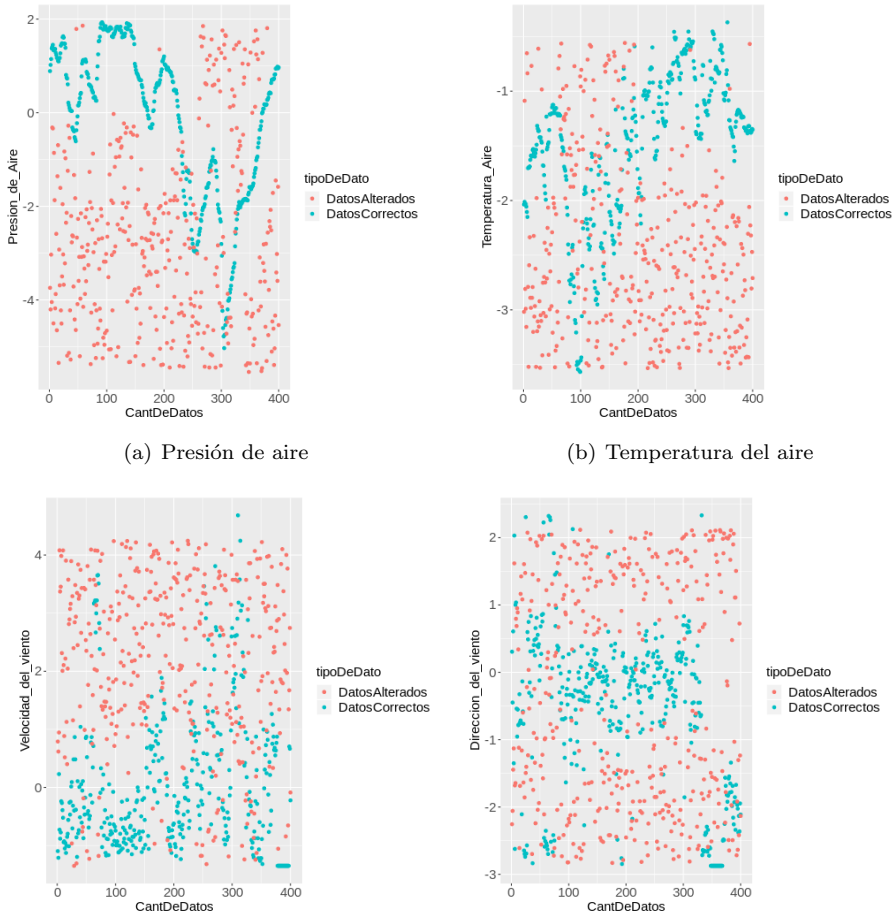


Figura 40: Tasa de detección de la variable errónea

Con este análisis, pareciera ser que el método separa bien los errores de los que no son errores en la mayoría de las variables. Para poder medir la precisión de nuestro método existen muchas técnicas, una de la más conocidas y la que usaremos en esta tesis es la curva ROC.

Antes de hacer el análisis de curva ROC, vamos a mostrar los gráficos de algunas variable, con 400 datos correctos y 400 datos alterados, para ver cuales son las diferencias entre ellos.



(a) Presión de aire

(b) Temperatura del aire

(c) Velocidad del viento

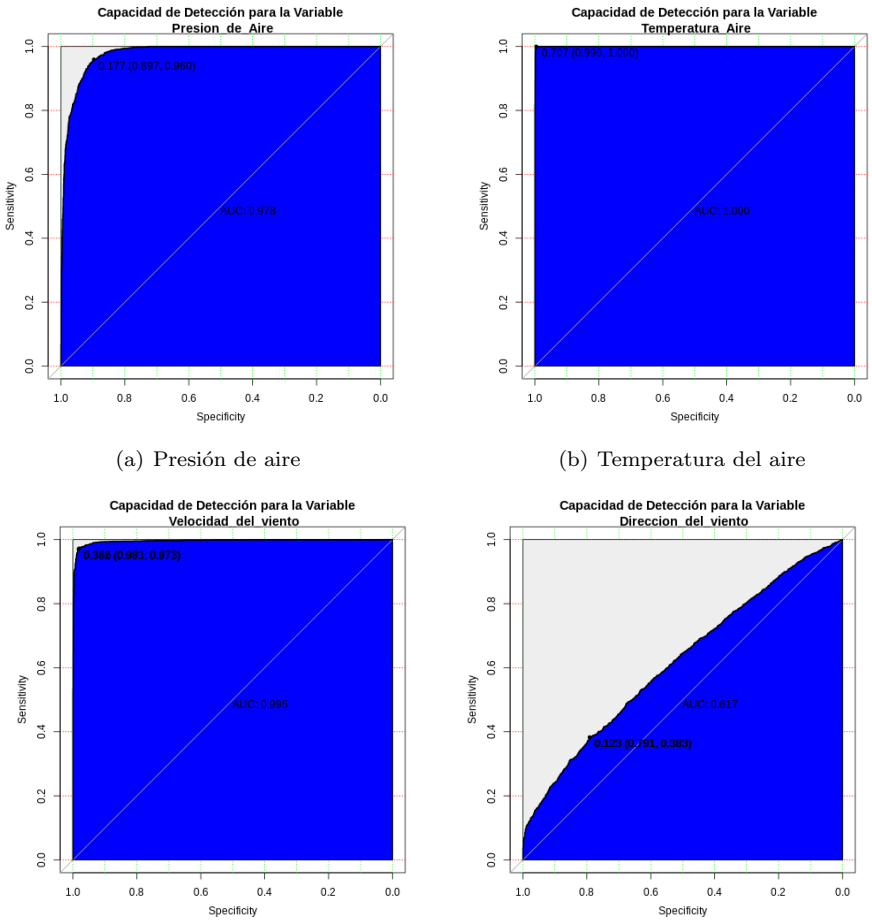
(d) Dirección del viento

Figura 41: Datos correctos y alterados

4.5.3. Análisis de curvas ROC

Para hacer el análisis de curvas ROC, tomamos una matriz de datos correctos y una matriz de datos alterados en la i-ésima columna, cada una de ellas de $\mathbb{R}^{2000\times 14}$. Lo primero que hacemos, es calcular las predicciones de cada una de ellas con nuestro "mejor" autoencoder y después calculamos el **mse** entre los datos predichos y los datos reales, por lo que vamos a obtener 2 listas, cada una con 2000 valores, que corresponderán a los **mse** de cada dato, uno esperaría que para los datos correctos, los errores fueran pequeños y que para los datos alterados fueran altos. Luego, con estas 2 listas haremos el análisis de curvas ROC. Este análisis lo vamos a hacer para cada matriz de datos alterados (14 veces), por lo que vamos a obtener 14 gráficos de área bajo la curva.

Los resultados que obtuvimos fueron muy buenos, ya que en casi todos los casos el área de la curva ROC se acercaba mucho a 1, a continuación mostramos los gráficos de algunas variables.



(a) Presión de aire

(b) Temperatura del aire

(c) Velocidad del viento

(d) Dirección del viento

Figura 42: Curvas ROC del MSE de la mejor, la peor variable y otras 2 elegidas al azar

4.5.4. Gráfico violín

Como pudimos ver en el análisis que realizamos anteriormente, entre mayor era la dimensión del espacio latente, mejor funcionaba nuestro autoencoder. Vamos a realizar una comparación de los MSE de los datos correctos y los erróneos de 2 variables(Temperatura y dirección del viento) usando gráficos de violín, para analizar como varían sus distribuciones. Elegimos estas 2 variables, ya que sabemos que nuestro autoencoder funciona bien para diferenciar los datos correctos de los erróneos para la variable temperatura pero no tanto para variable dirección del viento. En los siguientes gráficos veremos las distribuciones de los MSE de los datos erróneos y los correctos, para las distintas dimensiones de los espacios latentes. Lo que esperamos es que para el caso de la variable temperatura, las distribuciones se vayan separando a medida que aumente la dimensión del espacio latente y que no pase lo mismo en el caso de la variable dirección del viento. Terminamos graficando el logaritmo de los MSE, ya que ahí podíamos ver las diferencias más claramente.

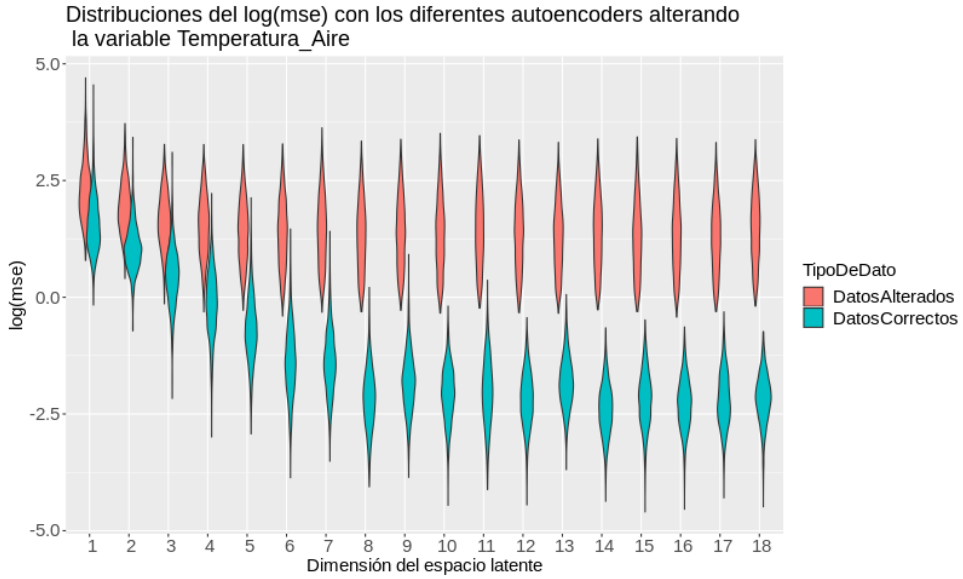


Figura 43: Distribuciones de los diferentes autoencoders para la variable Temperatura del Aire

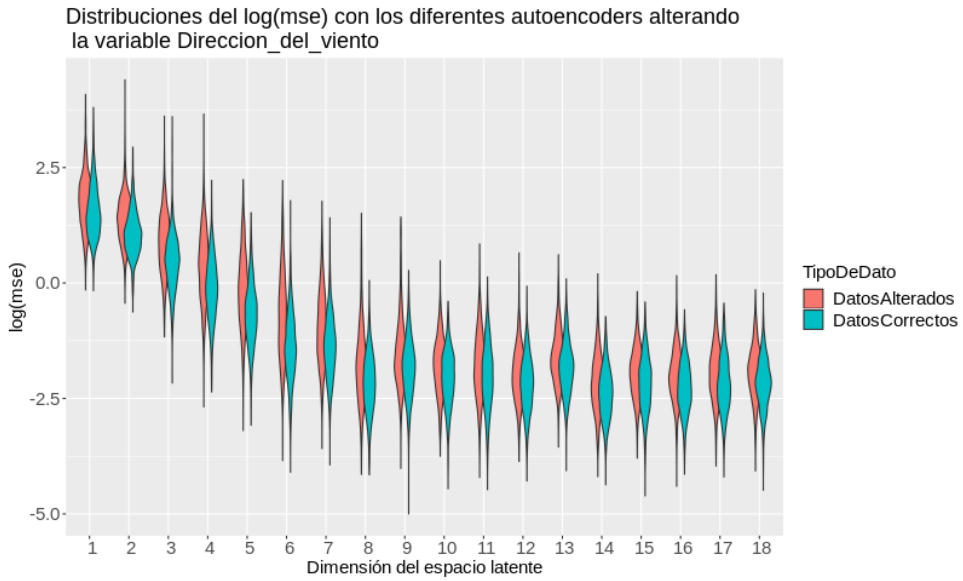


Figura 44: Distribuciones de los diferentes autoencoders para la variable Dirección del Viento

Luego de este análisis, podemos concluir que la elección del autoencoder con dimensión 8 en el espacio latente fue una buena, ya que vemos que los datos correctos están bien separados de los datos erróneos. El autoencoder de dimensión 7 tal vez podría haber funcionado bien también.

5. Aplicación de la metodología a una estación meteorológica en Argentina

5.1. Análisis descriptivo

Vamos a realizar un análisis similar al anterior, con otro conjunto de datos. Estos datos son de Las Compuertas, una localidad argentina ubicada en la provincia de Mendoza. En este set de datos, fueron grabadas 6 variables (humedad, temperatura, lluvia, radiación solar, dirección de viento, velocidad del viento) cada 15 minutos, durante más de 3 años entre 2014-2017. En este caso, como tenemos pocas variables,vamos a crear directamente las 4 variables relacionadas a los días y las horas para tener una mayor precisión en el análisis.

Con esto, vamos a tener un total de 10 variables:

Var1) Humedad.

Var2) Temperatura.

Var3) Lluvia.

Var4) Radiación Solar.

Var5) Dirección del viento.

Var6) Velocidad del viento

Var7) Hora Coordenada X

Var8) Hora Coordenada Y

Var9) Día Coordenada X

Var10) Día Coordenada Y

Pero, estas 4 variables nuevas, sólo las vamos a usar para crear nuestros autoencoders. Cuando alteremos las variables, vamos a hacerlo con las 6 que teníamos al principio, ya que a las variables que creamos no tiene sentido analizar los errores.

5.2. Control de consistencia

Con este set de datos, tuvimos un pequeño inconveniente al aplicar el control de consistencia ya que teníamos 2 variables cuyo **MAD** era 0 (Lluvia y radiación Solar). La forma de solucionar esto fue hacer el control de consistencia, pero sin tener en cuenta estas variables. Pero también es un problema trabajar con variables que tengan gran cantidad de ceros, por eso nuestra decisión fue usar una transformación que a cada dato, le da su probabilidad en la empírica, i.e el percentil que representa en la distribución empírica. Esto significa que, el dato más chico, tiene probabilidad 1/n, por lo tanto le va a asignar ese valor (donde n es la cantidad de datos), el valor más grande tendrá probabilidad 1 y el dato que está en la mitad tiene probabilidad 0.5 que es la mediana. Esta es una estandarización que nos devuelve valores entre 0 y 1, que además, nos permite mantener el orden de los valores.

5.3. Modelo

Este modelo es similar al que construimos para los datos de la estación Alemana, tiene una capa de entrada 7 capas intermedias y una capa de salida, la diferencia es que en este caso todas las capas intermedias usan la función de activación **ReLU** y disminuimos la cantidad de nodos de la tercera y la quinta capa ya que las dimensiones del set de datos eran menores

ENCODER:	**ESPACIO LATENTE:**	**DECODER:**
Capa de Entrada = 10 nodos		Quinta capa = 16 nodos
Primera capa = 128 nodos	Cuarta capa = i nodos $i \in \{1, ..., 10\}$	Sexta capa = 64 nodos
Segunda capa = 64 nodos		Séptima capa = 128 nodos
Tercera capa = 16 nodos		Capa de Salida = 10 nodos

5.4. Espacio latente

Al igual que con el set de datos de Jena, hacemos la elección del "mejor" autoencoder calculando los **Errores Medianos Absolutos** para cada dimensión del espacio latente. En este caso las matrices de los errores de reconstrucción para los datos test vivirán en $\mathbb{R}^{35918\times100}$

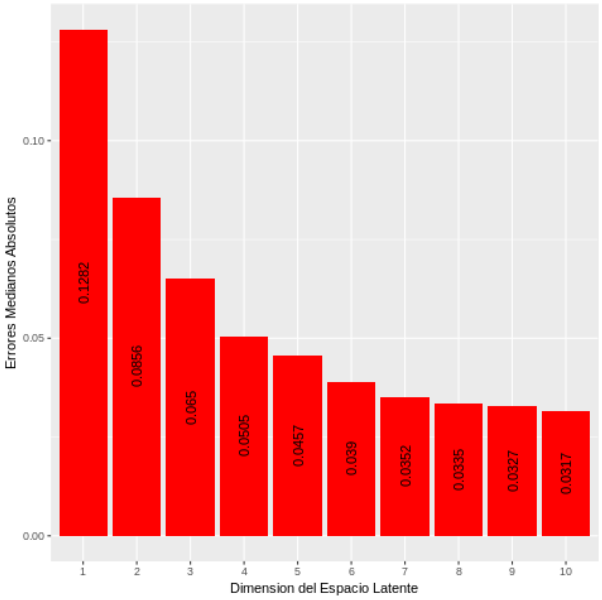
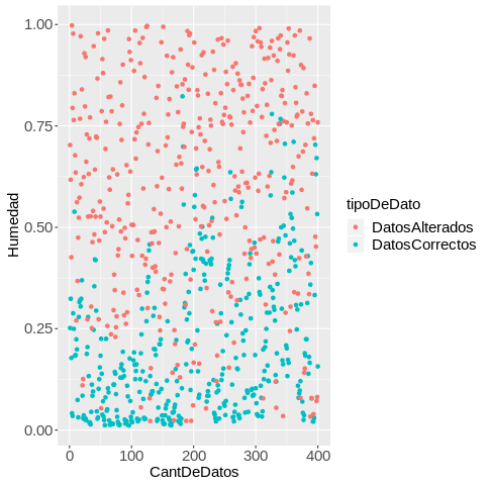


Figura 45: Errores Medianos Absolutos de cada uno de nuestros modelos

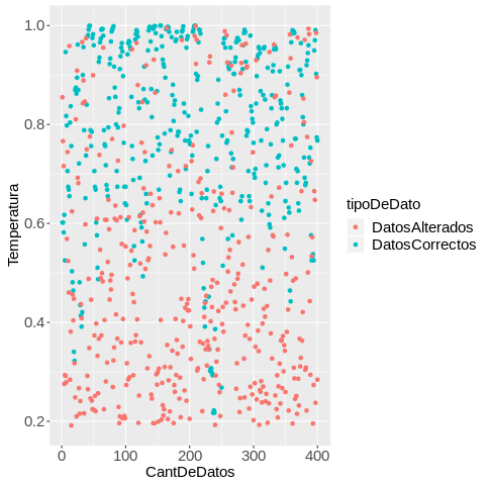
En el gráfico anterior, a medida que aumentamos la dimensión de nuestro espacio latente el **Error Mediano Absoluto** va disminuyendo y lo sigue haciendo hasta llegar a la dimensión del espacio total. En este caso decidimos quedarnos con el autoencoder con dimensión 7 en el espacio latente, ya que la diferencia entre los **Errores Medianos Absolutos** no es tan importante, menos de un 10 % entre las dimensiones 7 y 10 además tenemos una reducción de 3 dimensiones y el error sigue siendo bajo.

5.5. Análisis de datos correctos vs erróneos

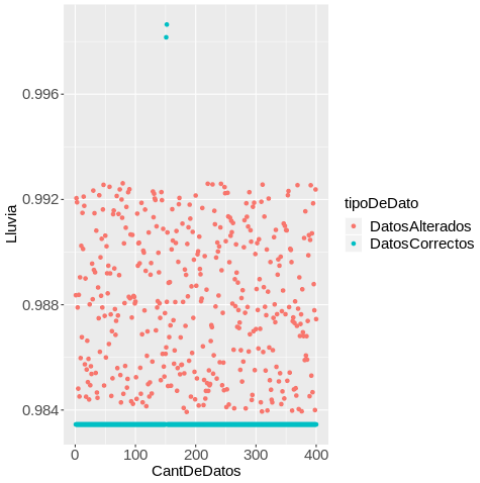
Al igual que hicimos con los datos de la estación Alemana, mostraremos algunos gráficos de las variables con los datos correctos y los datos erróneos, también el análisis de curvas ROC y el de la función variable errónea



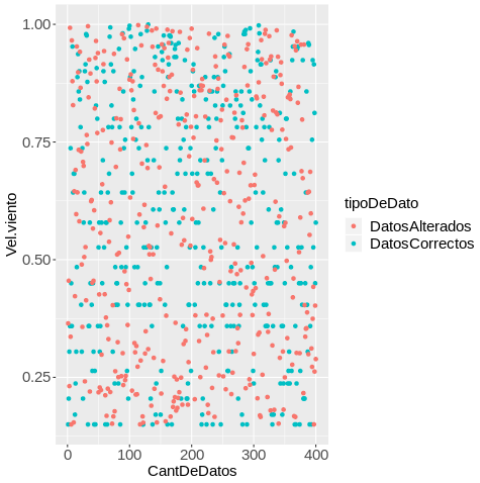
(a) Presión de aire



(b) Temperatura del aire



(c) Velocidad del viento



(d) Dirección del viento

Figura 46: Datos correctos y alterados

5.5.1. MAE y MSE

En este caso tomaremos 2 matrices de $\mathbb{R}^{2000\times 6}$ una con los datos correctos y la otra con la i-ésima columna alterada, luego les calculamos el **MAE** y el **MSE** con el autoencoder que construimos. Recordemos que este paso lo hacemos para cada una de las variables, por lo que en este caso tendremos 12 **MAE** y 12 **MSE** donde 6 de cada uno de ellos corresponde a los datos correctos y los otros 6 a los datos alterados.

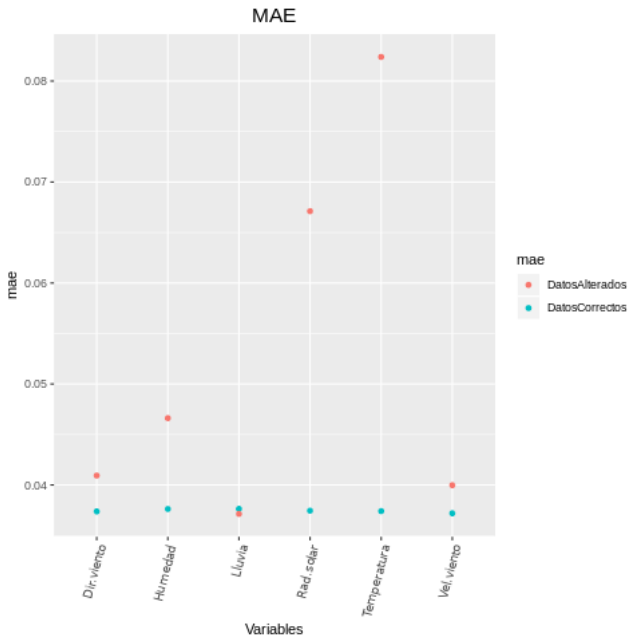


Figura 47: MAE de los datos correctos y alterados

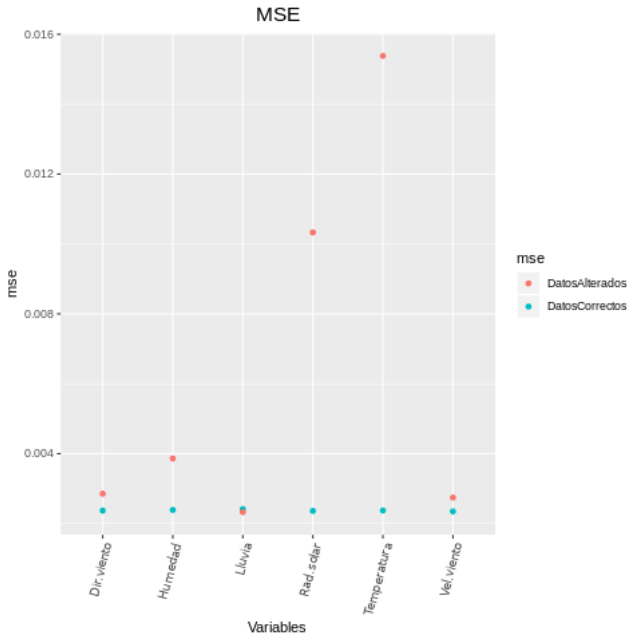
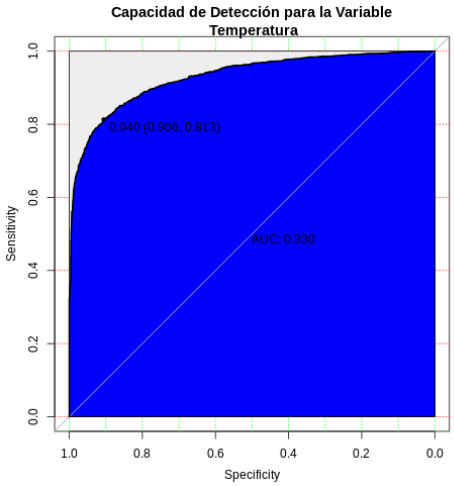


Figura 48: MSE de los datos correctos y alterados

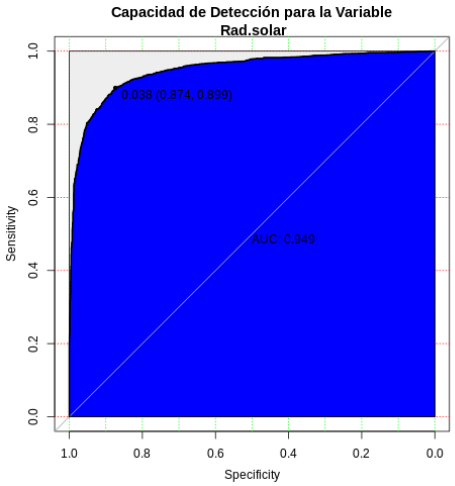
En estos gráficos podemos ver que hay varias variables donde los datos correctos y los datos erróneos no se diferencian demasiado, por lo que podemos llegar a esperar que los análisis de esas variables no sean muy prometedores.

5.5.2. Análisis de curvas ROC

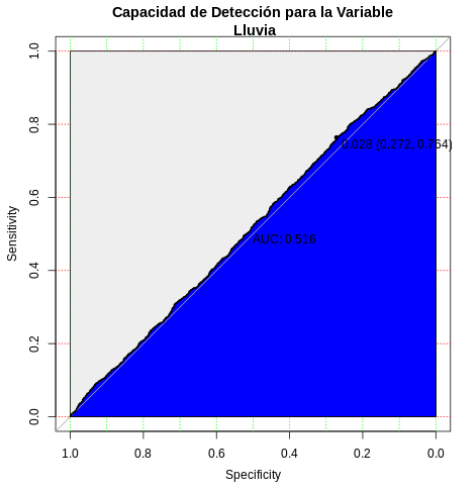
En este análisis vamos a usar las matrices con los datos correctos y con los datos alterados en la i-ésima columna, cada matriz es de $\mathbb{R}^{2000\times 6}$. Recordemos que lo que hacíamos en este caso es calcular las predicciones de cada una de las matrices con nuestro "mejor" autoencoder y después calculamos el **MSE** entre los datos predichos y los datos reales obteniendo en cada caso, una lista con 2000 valores, esperando que los datos correctos tengan errores bajos y los datos alterados tengan errores altos. El análisis de curva ROC lo aplicaremos a cada una de estas listas.



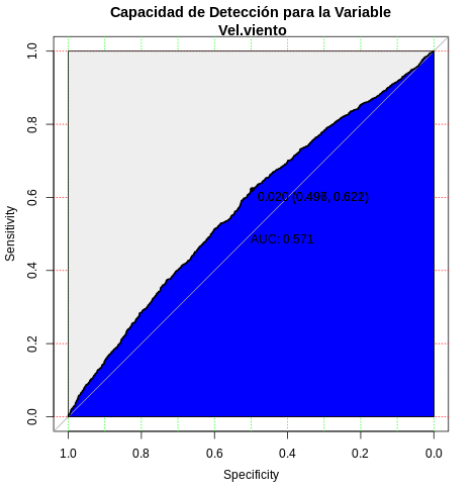
(a) Temperatura



(b) Radiación solar



(c) Lluvia



(d) Velocidad del viento

Figura 49: Curva ROC del MSE de las mejores y peores variables

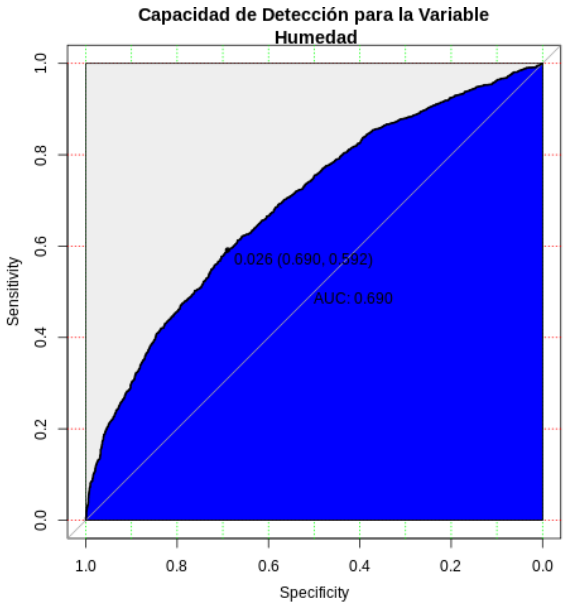


Figura 50: Curva ROC del MSE de la variable Humedad

En estos gráficos podemos ver que las variables Radiación solar y Temperatura son las que tienen mayor área bajo la curva, lo que significa que tienen buena capacidad discriminatoria y que las variables que están relacionadas con el viento y la lluvia, tienen muy mala capacidad discriminatoria, pero esto es esperable, ya que este tipo de variables son las más difíciles de predecir. Esto también pasaba con los datos Jena de la estación alemana; la variable que tenía peor área bajo la curva, era la dirección del viento.

Variables	Área bajo la curva
Lluvia	0.516
Velocidad del viento	0.571
Dirección del viento	0.586
Humedad	0.690
Temperatura	0.930
Radiación solar	0.949

Cuadro 1: Curva ROC

5.5.3. Método para detectar la variable que introduce el error

En esta sección lo que hacíamos era construir tantos autoencoders como variables a predecir tengamos (en este caso 6) y al autoencoder i, le quitamos la i-esima variable. Luego utilizando el mismo método que explicamos en la página 41 para los datos de Jena, que servía para identificar cual era la variable que introduce el error.

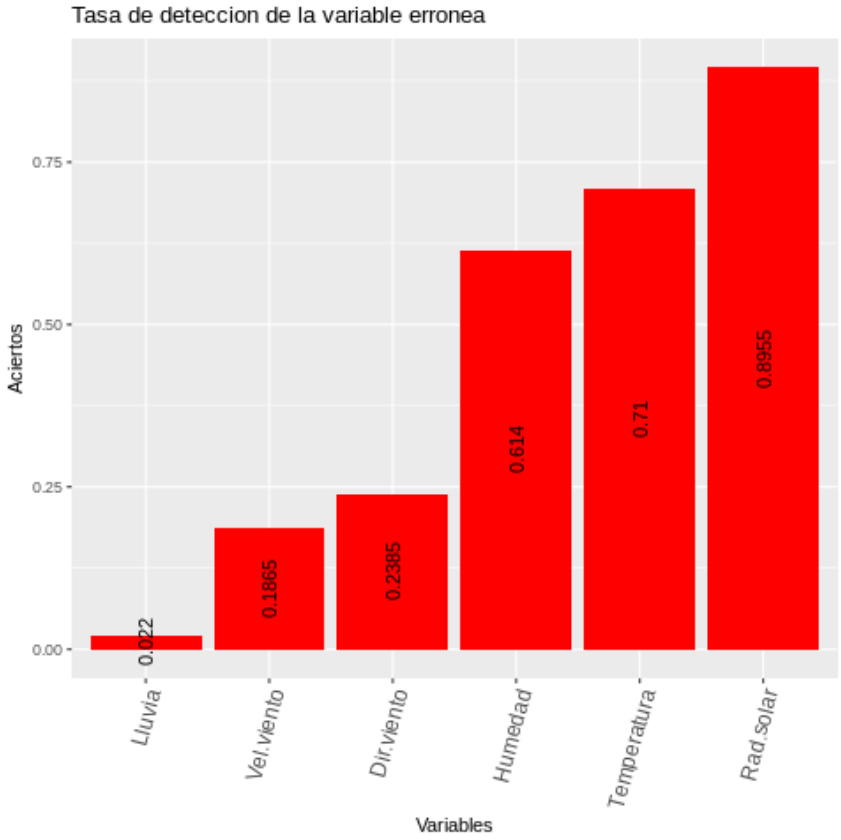


Figura 51: Tasa de detección de la variable errónea

En este caso, nuestra variable errónea, no funciona tan bien como antes, ya que como podemos observar en el gráfico de barras, en el caso que mejor detecta el error es en la variable Radiación Solar, con más de un 80 % de efectividad.

5.6. Variables rezagadas

Para tratar de mejorar la predicción de nuestro modelo, vamos a introducir las variables rezagadas. Lo que hacemos es incluir los valores pasados de las variables, es decir que si antes teníamos la variable temperatura a tiempo t_n, ahora también vamos a tenerla a tiempo t_{n-1} lo que nos va a dar un total de 16 variables.

Una vez hecho esto, volvimos a repetir el análisis para averiguar cual era la dimensión de espacio latente tenía nuestro "mejor" autoencoder y según el gráfico de los **Errores Medianos Absolutos** las mejores elecciones nos parecieron las de dimensión 11 y 12. Por lo que vamos a analizar los 2 casos.

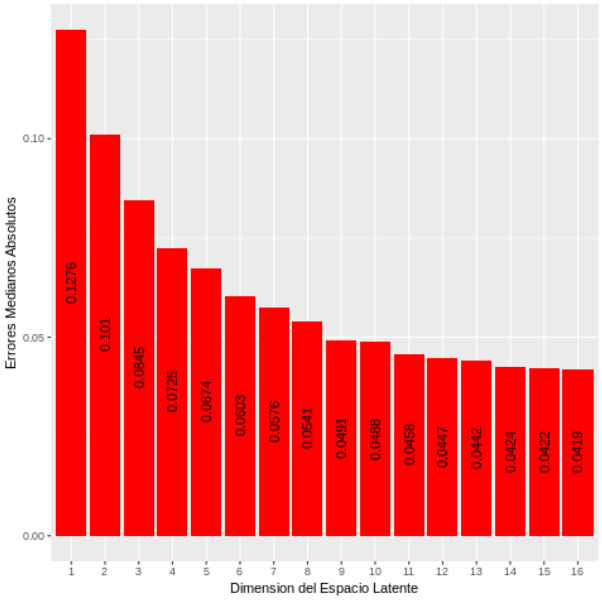


Figura 52: Errores Medianos Absolutos de cada uno de nuestros modelos

5.7. Variable rezagada con espacio latente de dimensión 11

En esta sección, repetiremos el análisis que hicimos anteriormente, pero para el autoencoder de dimensión 11 en el espacio latente.

5.7.1. MAE y MSE

Tanto para el **MAE** como para el **MSE** observamos que no hay una gran diferencia con el otro modelo, ya que las variables relacionadas con el viento y la lluvia tienen errores similares y las variables que se diferencian son Humedad, Radiación Solar y Temperatura.

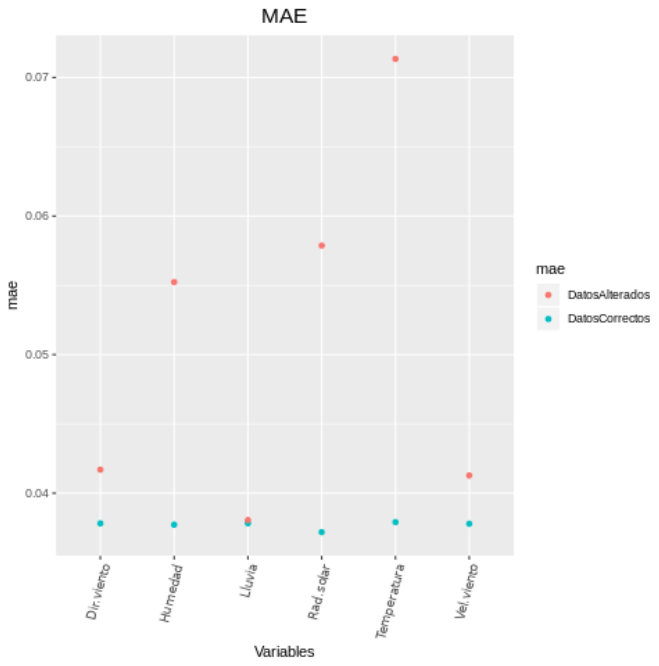


Figura 53: MAE de los datos correctos y alterados

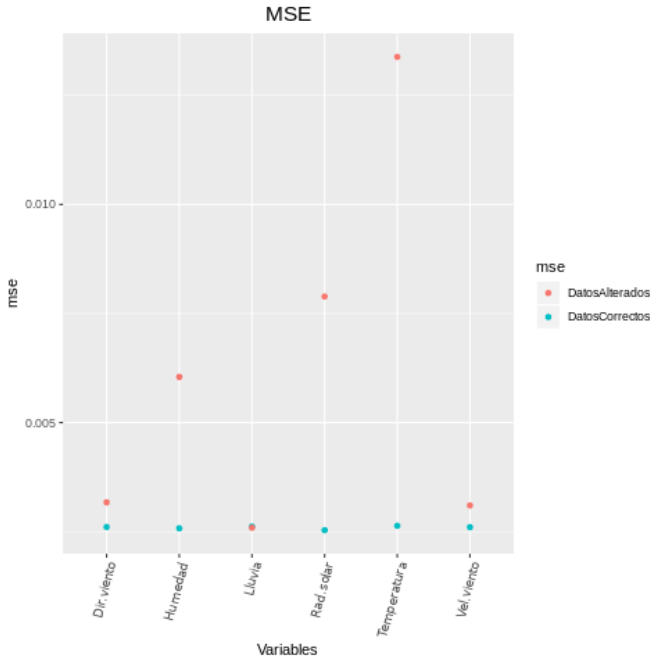
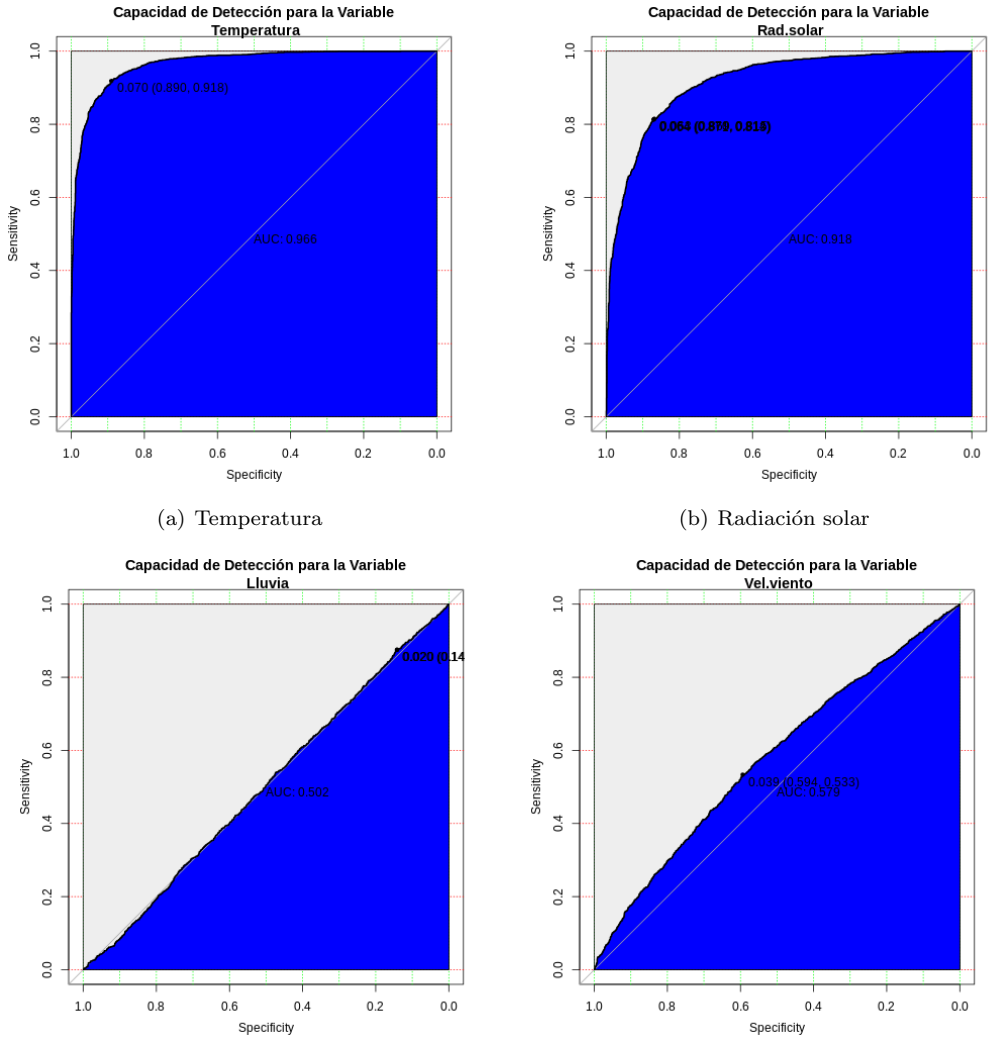


Figura 54: MSE de los datos correctos y alterados

5.7.2. Análisis de curvas ROC

En este caso, lo que pudimos observar fue que la variable Temperatura sigue teniendo una buena área bajo lo curva, pero hay una disminución en el caso de la Radiación Solar. Además las áreas de las variables Lluvia y Velocidad de Viento siguen siendo bajas, pero lo que notamos fue que hay una mejora en la variable Humedad, como podemos observar en los siguientes gráficos



(a) Temperatura

(b) Radiación solar

(c) Lluvia

(d) Velocidad del viento

Figura 55: Curva ROC del MSE de las mejores y peores variables

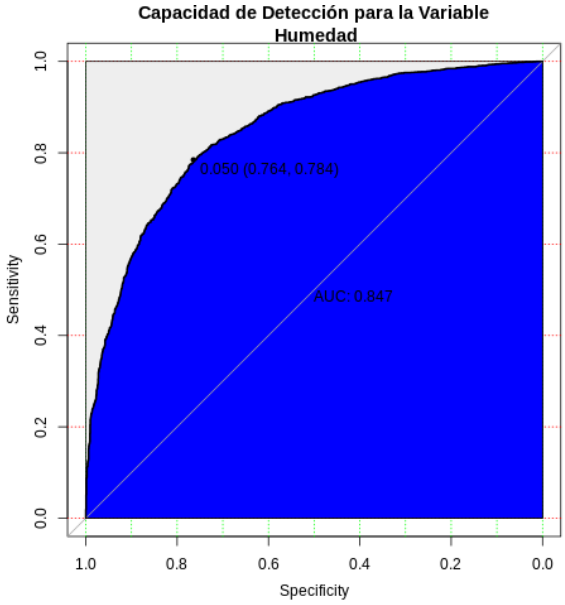


Figura 56: Curva ROC del MSE de la variable Humedad

Variables	Área bajo la curva
Lluvia	0.502
Velocidad del viento	0.579
Dirección del viento	0.594
Humedad	0.847
Radiación solar	0.918
Temperatura	0.966

Cuadro 2: Curva ROC

5.7.3. Método para detectar la variable que introduce el error

En este caso hay algunas variables que mejoraron su tasa de detección como Temperatura, Dirección del viento y Velocidad del viento. Como era de esperarse, la variable lluvia siguen siendo difícil de predecir, lo que si fue llamativo, es que la tasa de detección disminuyó para la variable Humedad y de forma importante para la variable Radiación Solar.

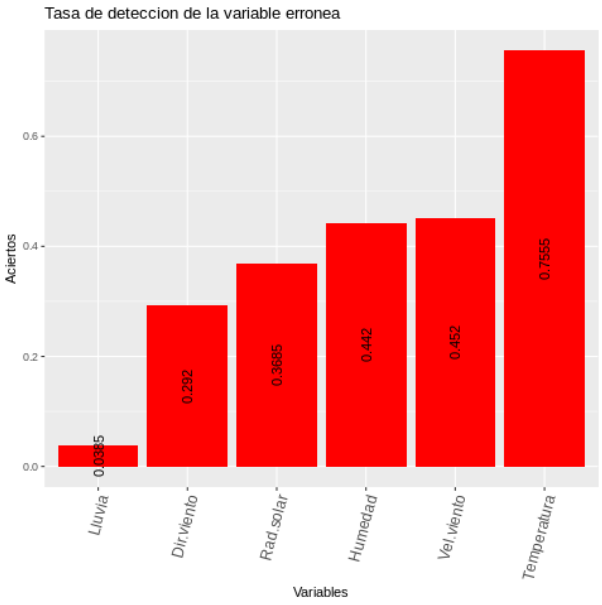


Figura 57: Tasa de detección de la variable errónea

5.8. Variable rezagada con espacio latente de dimensión 12

En este caso haremos el mismo análisis pero con el autoencoder de dimensión 12 en el espacio latente y compararemos los resultados entre los diferentes autoencoders

5.8.1. MAE y MSE

Al igual que con el autoencoder de dimensión 11 en el espacio latente, vemos que las variables que se diferencian son la Humedad, Temperatura, Radiación Solar y que para la lluvia y las relacionadas con el viento hay muy poco diferencia entre los errores.

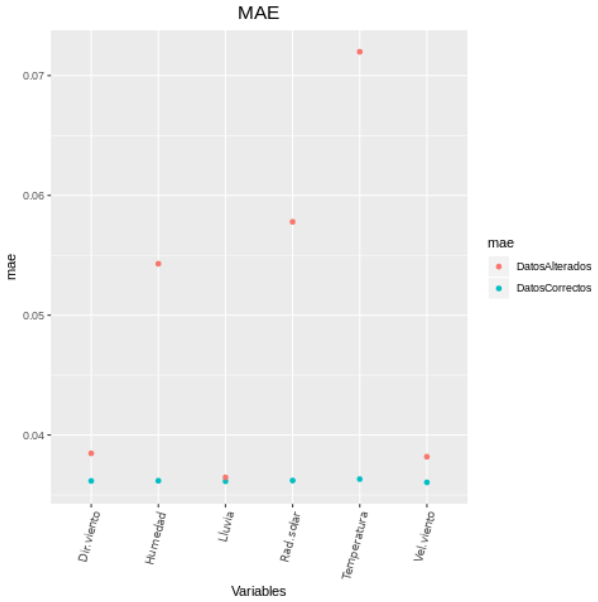


Figura 58: MAE de los datos correctos y alterados

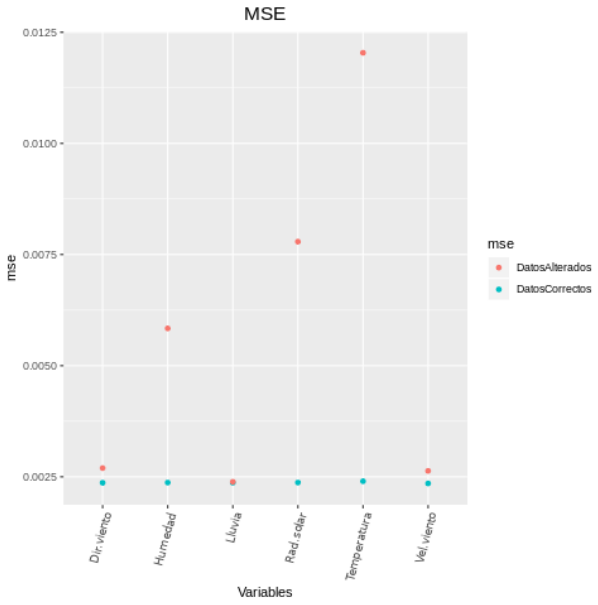
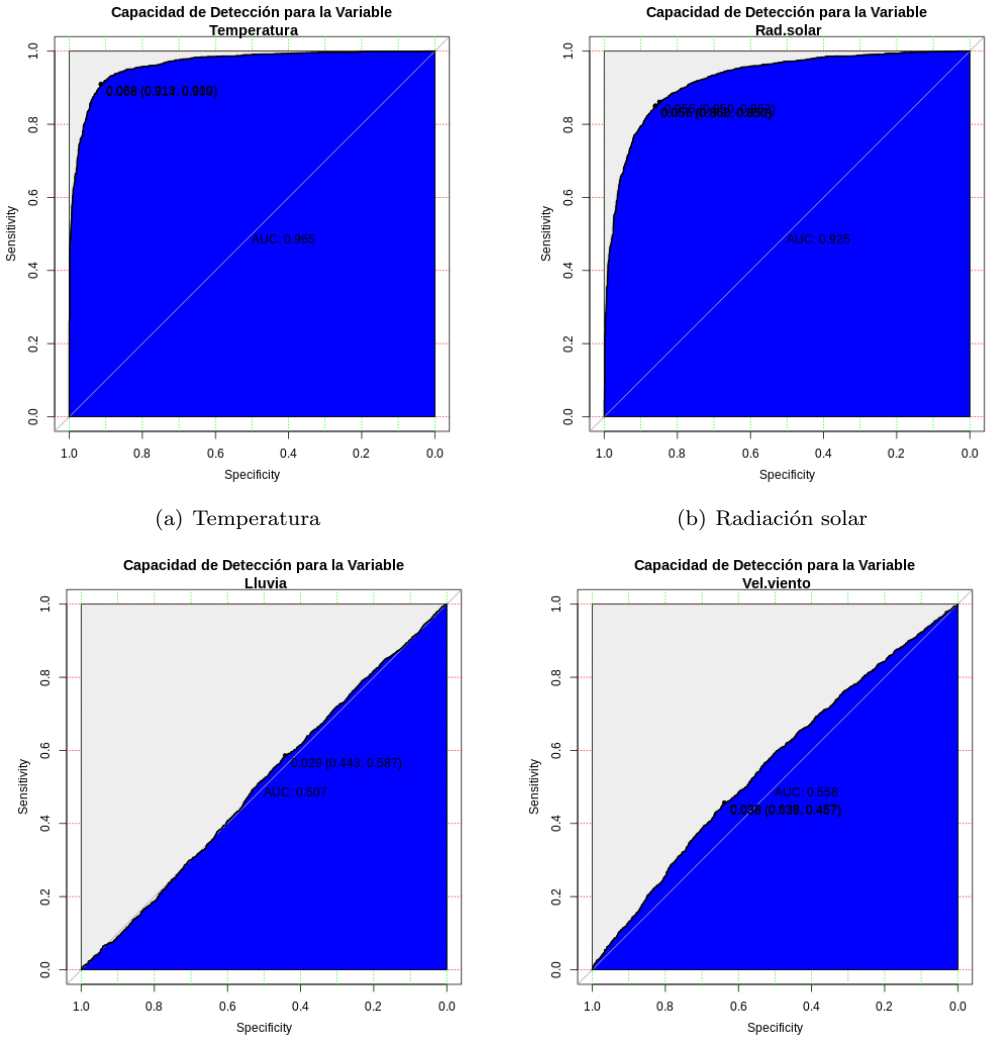


Figura 59: MSE de los datos correctos y alterados

5.8.2. Análisis de curvas ROC

En el análisis de curva ROC, tampoco notamos ninguna diferencia significativa, los resultados de todos los gráficos son muy similares.



(a) Temperatura

(b) Radiación solar

(c) Lluvia

(d) Velocidad del viento

Figura 60: Curva ROC del MSE de las mejores y peores variables

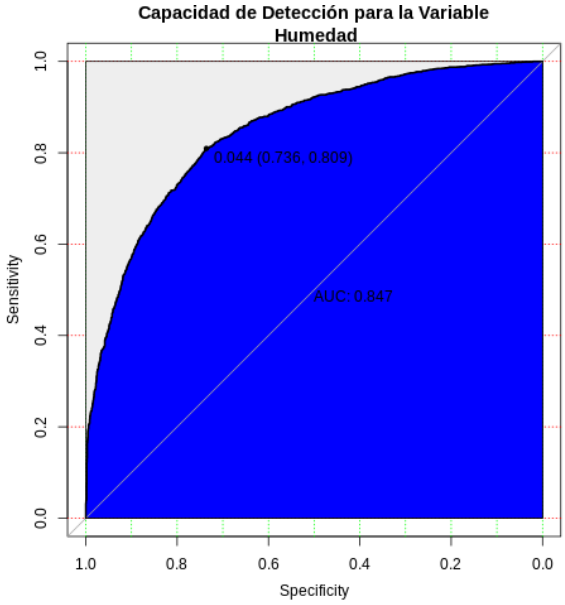


Figura 61: Curva ROC del MSE de la variable Humedad

Variables	Área bajo la curva
Lluvia	0.507
Velocidad del viento	0.558
Dirección del viento	0.570
Humedad	0.847
Radiación solar	0.925
Temperatura	0.966

Cuadro 3: Curva ROC

5.8.3. Método para detectar la variable que introduce el error

En este análisis si pudimos notar una gran diferencia con lo que vimos antes en el caso del autoencoder de dimensión 11 en el espacio latente, si los comparamos, podemos ver que mejoró la tasa de detección para las variables Temperatura, Humedad y Radiación Solar, pero disminuyó para las variables relacionadas con el viento.

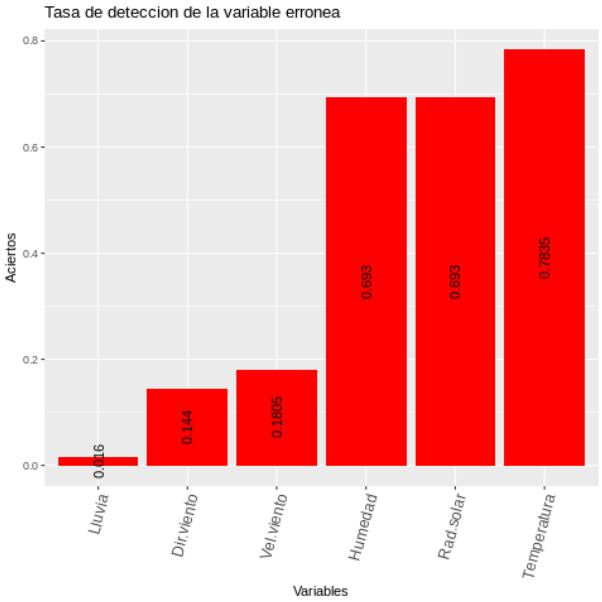


Figura 62

Ahora, cuando comparamos este modelo de variables rezagadas con nuestro modelo original, vemos que a pesar de que tiene más variables, no hay una gran mejoría respecto en la tasa de detección de la variable errónea, ya que solo vemos un aumento en la variable humedad y en el resto de los casos, disminuye. Por lo tanto, en este caso consideramos que el modelo de variables rezagadas no nos fue útil para mejorar nuestro modelo.

5.9. Gráfico Violín

Para este tipo de gráficos recordemos que lo que hacíamos era comparar los logaritmos de los MSE de los datos correctos y los alterados para 2 variables, en este caso también usaremos temperatura y dirección del viento. Los modelos de los autoencoders, son los mismos que usamos antes, lo único que hacemos es ir modificando la dimensión en el espacio latente.

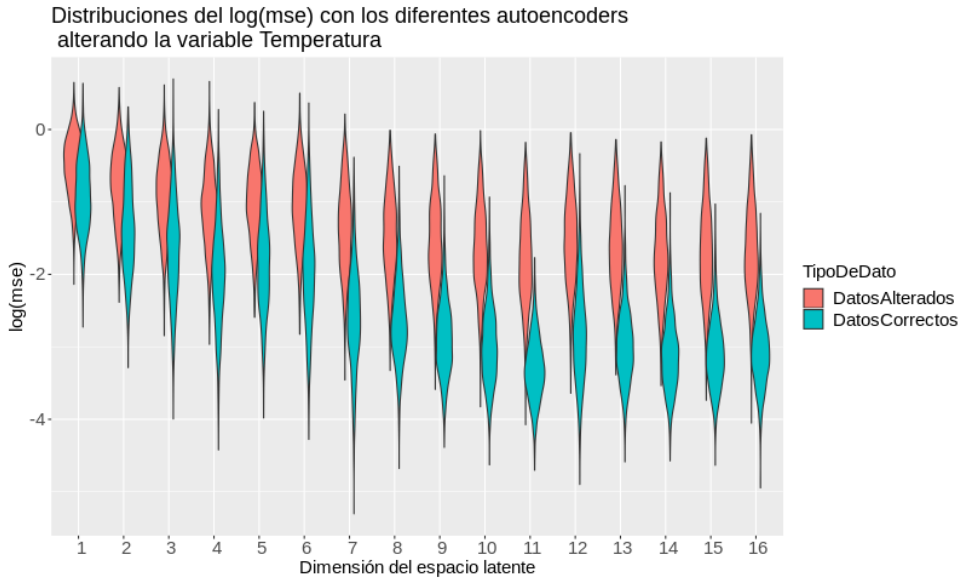


Figura 63: Distribuciones de los diferentes autoencoders para la variable Temperatura

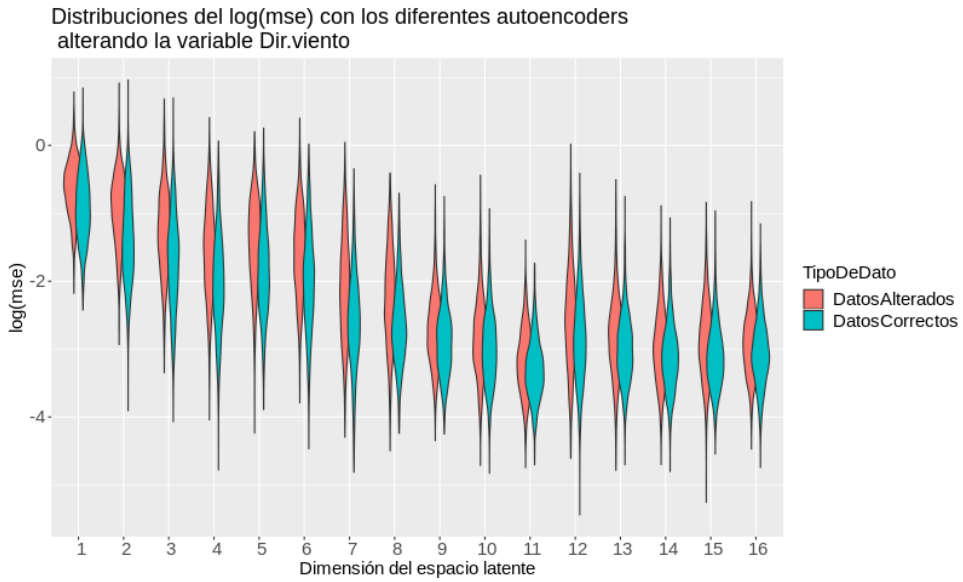


Figura 64: Distribuciones de los diferentes autoencoders para la variable Dirección del viento

Para el caso de la variable dirección del viento, era esperable que las distribuciones no se separaran ya que modelo no la predecía bien, pero para el caso temperatura, vemos como a medida que aumenta la dimensión del espacio latente, las distribuciones se separan más y podemos ver que a partir modelo con espacio latente de dimensión 7 ya se empiezan a diferenciar mejor.

6. Conclusiones

El propósito de esta tesis era crear un modelo de deep learning, basado en autoencoder para la detección de errores en datos meteorológicos.

En primer lugar se trabajó con el conjunto de datos de la estación alemana en Jena a los cuales se les aplicó un control de consistencia para eliminar outliers. Con este set de datos "limpios"se corrió varias veces el modelo para determinar cual era el "mejor" espacio latente que le podía corresponder. Se pudo observar que el "mejor" espacio latente en promedio, fue el que tenía dimensión 12, pero también notamos que el autoencoder con dimensión 8 ganaba en muchas ocasiones, por lo que decidimos analizar los 2 casos, ya que cuando uno trabaja con autoencoders lo que quiere es reducir la dimensionalidad de los datos. Se tomó un subconjunto del conjunto de datos test y se los alteró para probar cual era el poder de detección de errores en nuestros modelos. Creamos un método para identificar cual era la variable responsable del error y utilizamos herramientas para el análisis descriptivo como la **curva ROC** y los **gráficos violín**. Después de completar el análisis, concluimos que el autoencoder con dimensión 8 en el espacio latente era el mejor para la detección de errores, en este caso logramos una gran reducción de la dimensión ya que partimos con un modelo de dimensión 18.

Datos Jena Autoencoder dimensión 8		
Variable	Área bajo la curva	Porcentaje de Aciertos
Presión de Aire en milibar	0.978	82,60 %
Temperatura del Aire en grados Celcius	1.0	99,9 %
Temperatura Potencial en grados Kelvin	1.0	99,9 %
Temperatura de Rocio	1.0	99,9 %
Porcentaje de Humedad Relativa	0.999	98,30 %
Presión de vapor de saturación	1.0	99,95 %
Presión de Vapor	1.0	93,35 %
Déficit del Presión de Vapor	1.0	99,95 %
Humedad Específica	0.999	96,90 %
Concentración de vapor de agua	0.999	96,25 %
Densidad del Aire	1.0	97,6 %
Velocidad del viento	0.995	63,90 %
Máxima velocidad el aire	0.995	48,90 %
Dirección del viento en grados	0.617	64,60 %

Cuadro 4: Porcentaje de aciertos y área bajo la curva de los datos de Jena

En segundo lugar, repetimos nuestro análisis para otro conjunto de datos de la localidad de **Las Compuertas** en la provincia de Mendoza. En este caso, el "mejor" autoencoder es el que tiene dimensión latente igual a 7. Para este conjunto de datos la función variable errónea no funcionó tan bien como para el conjunto de datos de Jena ya que obtuvimos 3 variable que estaban por encima del 60 % en la tasa de detección de la variable errónea. Esto era esperable, ya que las curvas ROC nos mostraban 3 gráficos donde el modelo parecía funcionar bien.

Por último, para intentar mejorar nuestro modelo, utilizamos la técnica de la variable rezagada. En este caso realizamos los análisis para 2 modelos de autoencoder, uno con dimensión 11 en el espacio latente y otro con dimensión 12. Para el caso de dimensión 11, consideramos que el modelo empeoró, ya que aunque agregamos nuevas variables, sólo obtuvimos una sola variable cuya tasa de detección de la variable errónea estuviera por encima del 60 %. En el caso de dimensión 12, vimos que era mejor que el de dimensión 11, pero cuando fuimos a compararlo con nuestro modelo original, nos dimos cuenta que la tasa de detección de la variable errónea sólo mejoraba para la variable humedad y en el resto de las variables disminuía, por eso decidimos que para nuestro caso, la técnica de la variables rezagadas, no nos fue de utilidad para mejorar nuestro modelo, aún así es una muy buena técnica que se debe tener en cuenta para cuando se hace detección de errores en datos meteorológicos.

La conclusión final es que no existe un modelo perfecto en la detección de errores, sino que cada set de datos es único y tiene sus propias complicaciones, pero lo que podemos hacer es usar el conocimiento que obtenemos de cada uno de ellos para poder construir un modelo que nos ayude en cada caso.

Datos Las Compuertas Autoencoder dimensión 7		
Variable	Área bajo la curva	Porcentaje de Aciertos
Humedad	0.690	61,4 %
Temperatura	0.930	71 %
Lluvia	0.516	2.2 %
Radiación Solar	0.949	89,55 %
Dirección del viento	0.586	23,85 %
Velocidad del viento	0.571	18,65 %

Cuadro 5: Porcentaje de aciertos y área bajo la curva de los datos de Las Compuertas

Datos Las Compuertas Autoencoder dimensión 11		
Variable	Área bajo la curva	Porcentaje de Aciertos
Humedad	0.847	44,2 %
Temperatura	0.966	75,55 %
Lluvia	0.502	3.85 %
Radiación Solar	0.918	36,85 %
Dirección del viento	0.594	29,2 %
Velocidad del viento	0.579	45,2 %

Cuadro 6: Porcentaje de aciertos y área bajo la curva de los datos de Las Compuertas

Datos Las Compuertas Autoencoder dimensión 12		
Variable	Área bajo la curva	Porcentaje de Aciertos
Humedad	0.847	69,3 %
Temperatura	0.966	69,3 %
Lluvia	0.507	1.6 %
Radiación Solar	0.925	78,35 %
Dirección del viento	0.570	13,4 %
Velocidad del viento	0.558	18,5 %

Cuadro 7: Porcentaje de aciertos y área bajo la curva de los datos de Las Compuertas

7. Bibliografía

Referencias

[1] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386, 1958.

[2] W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics, 5(4), pp. 115–133, 1943

[3] C. M. Bishop. Neural networks for pattern recognition. Oxford University Press, 1995.

[4] S. Gallant. Perceptron-based learning algorithms. IEEE Transactions on Neural Networks, 1(2), pp. 179–191, 1990.

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016): Deep Learning. MIT Press

Aggarwal, Charu C - Neural Networks and Deep Learning

https://www.manning.com/books/deep-learning-with-r