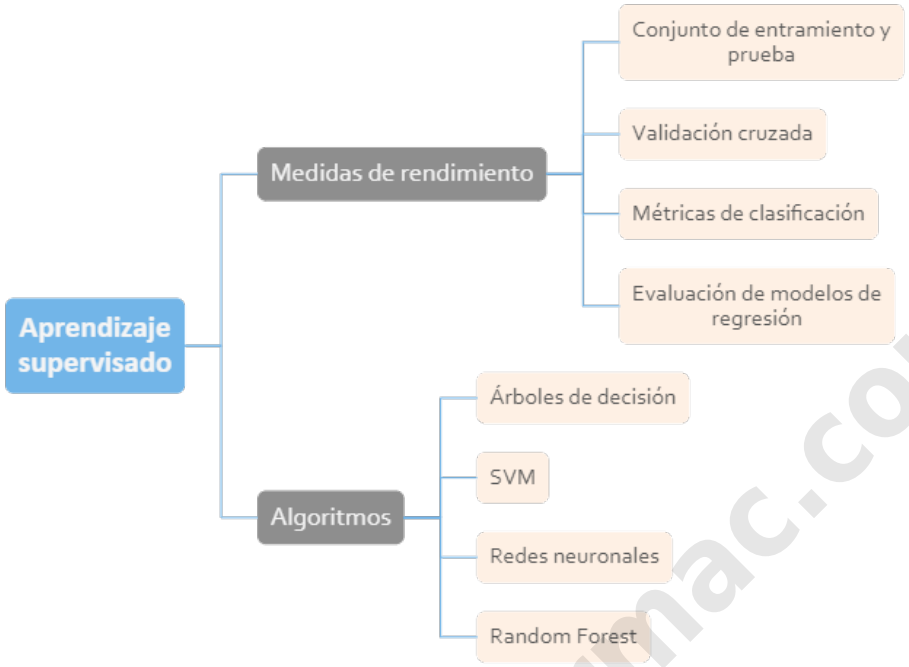
Introducción

El **aprendizaje supervisado** generalmente se realiza en el **contexto** de la clasificación, cuando queremos mapear la entrada a las etiquetas de salida, o la regresión, cuando queremos mapear la entrada a una salida continua.

Objetivos

- Conocer el aprendizaje supervisado.
- Estudiar las aplicaciones del aprendizaje supervisado.
- Comparar las medidas de rendimiento del aprendizaje supervisado.
- Aprender a desarrollar modelos lineales.
- Estudiar algoritmos de árboles, SVM y redes neuronales.
- Valorar la implementación de Random Forest.

Mapa Conceptual



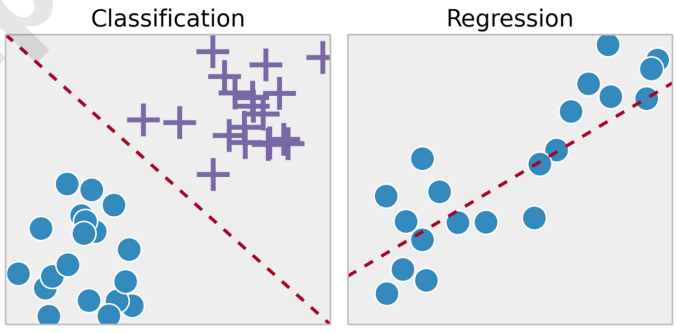
Definición y aplicaciones.

En el **aprendizaje supervisado**, la computadora aprende a través del ejemplo. Aprende de datos pasados y aplica el aprendizaje a los datos presentes para predecir eventos futuros. En este caso, tanto los datos de entrada como los datos de salida deseados proporcionan ayuda para la predicción de eventos futuros.

Para predicciones precisas, los datos de entrada se etiquetan como la respuesta correcta.

Es importante recordar que todos los algoritmos de aprendizaje supervisado son esencialmente algoritmos complejos, en modelos de clasificación o regresión.

- **Modelos de clasificación**: los modelos de clasificación se utilizan para problemas en los que se puede categorizar la variable de salida, como "Sí" o "No", o "Pasar" o "Fallar". Los modelos de clasificación se utilizan para predecir la categoría de los datos. Los ejemplos de la vida real incluyen detección de spam, análisis de sentimientos, predicción de exámenes, etc.
- **Modelos de regresión**: los modelos de regresión se utilizan para problemas en los que la variable de salida es un valor real, como un número único, dinero, salario, peso o presión. Se suele utilizar para predecir valores numéricos basados en observaciones de datos anteriores. Algunos de los algoritmos de regresión más populares incluyen regresión lineal, lineal múltiple, polinómica, de vector de soporte, árbol de decisión y random forest.



Clasificación frente a regresión

Los **algoritmos de aprendizaje automático supervisado** definen modelos que capturan las relaciones entre los datos. La clasificación es un área de aprendizaje automático supervisado que

intenta predecir a qué clase o categoría pertenece una entidad, en función de sus características.

Por ejemplo, podríamos analizar a los empleados de alguna empresa e intentar establecer una dependencia sobre las características o variables, como el nivel de educación, el número de años en un puesto actual, la edad, el salario, las probabilidades de ser ascendido, etc. El conjunto de datos relacionados con un solo empleado es una observación. Las características o variables pueden tomar una de dos formas:

- Las variables independientes, también llamadas entradas o predictores, no dependen de otras características de interés (o al menos se asume así a efectos del análisis).
- Las variables dependientes, también llamadas salidas o respuestas, dependen de las variables independientes.

En el ejemplo anterior, donde está analizando a los empleados, se puede suponer que el nivel de educación, el tiempo en una posición actual y la edad son mutuamente independientes, y considerarlos como las entradas. El salario y las probabilidades de ascenso podrían ser las salidas que dependen de las entradas.

Los **algoritmos de aprendizaje automático supervisado** analizan una serie de observaciones y tratan de expresar matemáticamente la dependencia entre las entradas y salidas. Estas representaciones matemáticas de dependencias son los modelos.

La naturaleza de las variables dependientes diferencia los problemas de regresión y clasificación. Los problemas de regresión tienen resultados continuos y generalmente no acotados. Un ejemplo es cuando estamos estimando el salario en función de la experiencia y el nivel de educación. Por otro lado, los problemas de clasificación tienen salidas discretas y finitas llamadas clases o categorías. Por ejemplo, predecir si un empleado va a ser promovido o no (verdadero o falso) es un problema de clasificación.

Hay dos tipos principales de problemas de clasificación:

- Clasificación binaria o binomial: solamente dos clases para elegir (generalmente 0 y 1, verdadero y falso, o positivo y negativo).
- Clasificación multiclase o multinomial: tres o más clases de las salidas a elegir.

Si solo hay una variable de entrada, generalmente se denota con x. Para más de una entrada, comúnmente se utilizará la notación vectorial x = (x_1, ..., xr), donde r es el número de los predictores (o características independientes). La variable de salida a menudo se denota con y, y toma los valores 0 o 1.

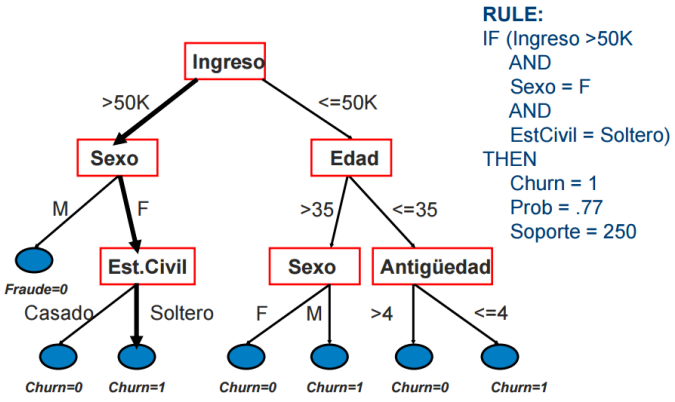
Hay algunas aplicaciones muy prácticas de los **algoritmos de aprendizaje supervisado** en la vida real, como las siguientes:

- Categorización de texto.
- Detección de rostros.
- Reconocimiento de firmas.
- Descubrimiento de clientes.
- Detección de spam.
- Pronóstico del tiempo.
- Predicción de los precios de la vivienda en función del precio de mercado prevaleciente.
- Predicciones del precio de las acciones.

A continuación, vemos brevemente algunos algoritmos de aprendizaje supervisado.

Árboles de decisión

Un árbol de decisión es una herramienta de apoyo a la decisión que utiliza un gráfico o modelo de decisiones en forma de árbol y sus posibles consecuencias, incluidos los resultados de eventos fortuitos, los costes de recursos y la utilidad.



Desde el punto de vista de la decisión comercial, un árbol de decisión es el número mínimo de preguntas de sí / no que uno tiene que hacer, para evaluar la probabilidad de tomar una decisión correcta (en la mayoría de los casos). Como método, nos permite abordar el problema de una manera estructurada y sistemática para llegar a una conclusión lógica.

Clasificación naive (ingenua) de Bayes.

Los clasificadores ingenuos de Bayes son una familia de clasificadores probabilísticos simples basados en la aplicación del teorema de Bayes con fuertes supuestos de independencia (ingenuos) entre las características. La imagen presentada es la ecuación: con P (A | B) es la probabilidad posterior, P (B | A) es la probabilidad, P (A) es la probabilidad previa de clase y P (B) es la probabilidad previa del predictor.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

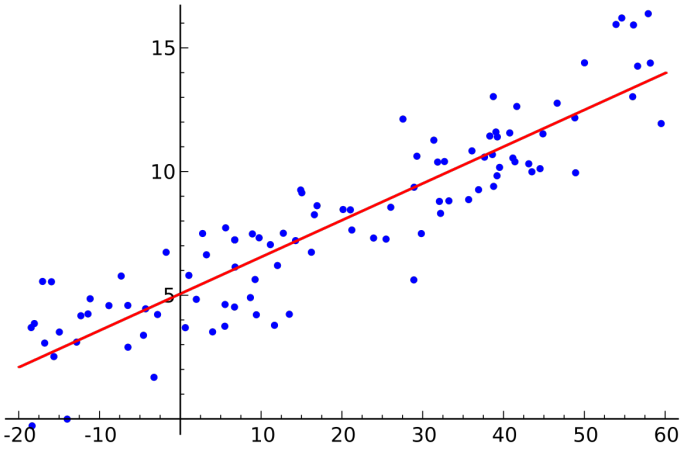
Algunos ejemplos del mundo real son:

- Marcar un correo electrónico como spam.
- Clasificar un artículo de noticias sobre tecnología, política o deportes.
- Verificar si un texto expresa emociones positivas o negativas.
- Utilizado para el software de reconocimiento facial.

Regresión de mínimos cuadrados ordinarios

Mínimos cuadrados es un método para realizar regresión lineal. Podemos pensar en la regresión lineal como la tarea de ajustar una línea recta a través de un conjunto de puntos. Hay varias estrategias posibles para hacer esto.

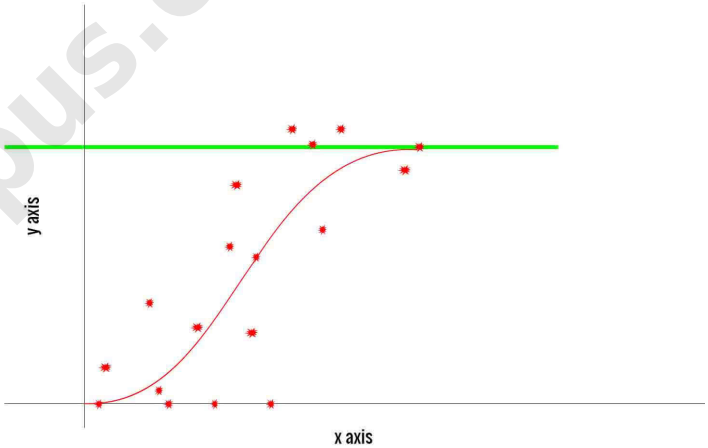
La **estrategia de "mínimos cuadrados ordinarios"** es la siguiente: dibujamos una línea y luego, para cada uno de los puntos de datos, medimos la distancia vertical entre el punto y la línea, y procedemos a sumarlos. La línea ajustada sería aquella donde esta suma de distancias es lo más pequeña posible.



Lineal se refiere al tipo de modelo que se está utilizando para ajustar los datos, mientras que los mínimos cuadrados se refieren al tipo de métrica de error que está minimizando.

Regresión logística

Es una forma estadística muy potente para modelar un resultado binomial con una o más variables explicativas. Mide la relación entre la variable dependiente categórica y una o más variables independientes mediante la estimación de probabilidades utilizando una función logística, que es la distribución logística acumulativa.



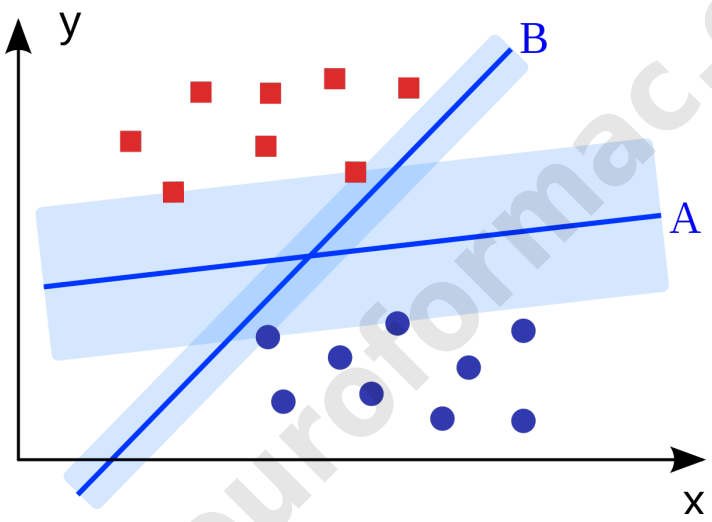
En general, las regresiones se pueden usar en aplicaciones del mundo real como:

- Puntuación de crédito.
- Medir las tasas de éxito de las campañas de marketing.
- Predecir los ingresos de un determinado producto.

- Evaluar si se producirá un terremoto en un día en particular.

Máquinas de vectores de soporte (Support Vector Machine)

Es un algoritmo de clasificación binario. Dado un conjunto de puntos de 2 tipos en lugar de N dimensiones, SVM genera un hiperplano dimensional (N - 1) para separar esos puntos en 2 grupos. Digamos que tiene algunos puntos de 2 tipos en un documento que son linealmente separables. SVM encontrará una línea recta que separa esos puntos en 2 tipos y está situada lo más lejos posible de todos esos puntos.



En términos de escala, algunos de los mayores problemas que se han resuelto utilizando SVM son la publicidad gráfica, el reconocimiento de “splice site” humano en genética, detección de género basada en imágenes, clasificación de imágenes a gran escala, etc.

Métodos de ensamblaje

Son algoritmos de aprendizaje que construyen un conjunto de clasificadores y luego clasifican nuevos datos con puntos mediante un voto ponderado de sus predicciones. El método de conjunto original es el promedio Bayesiano, pero los algoritmos más recientes incluyen la codificación de salida de corrección de errores, el embolsado y el refuerzo.

Los métodos de ensamblaje aportan una serie de ventajas frente a los modelos individuales:

- Promueven sesgos.

- Reducen la varianza: la opinión agregada de un grupo de modelos es menos ruidosa que la opinión individual de uno de los modelos. En finanzas, esto se llama diversificación: una cartera mixta de muchas acciones será mucho menos variable que solo uno de los valores. Es por eso que sus modelos serán mejores con más puntos de datos en lugar de menos.
- Es poco probable que se ajusten demasiado.

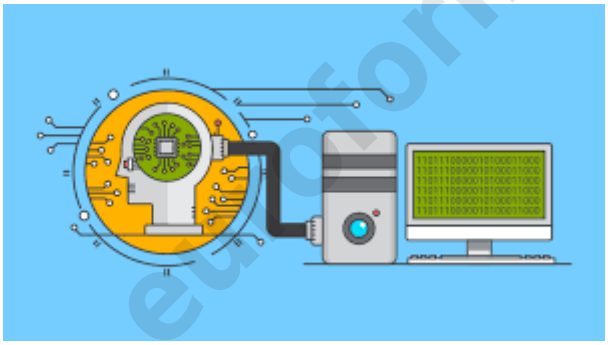
Los algoritmos de aprendizaje automático supervisado definen modelos que capturan las relaciones entre los datos. La clasificación es un área de aprendizaje automático supervisado que intenta predecir a qué clase o categoría pertenece una entidad, en función de sus características.

Verdadero.
Falso.

Medidas de rendimiento.

La **evaluación de modelos** (incluida la evaluación de modelos de aprendizaje supervisados y no supervisados) es el proceso de medir objetivamente cómo de bien los modelos de aprendizaje automático realizan las tareas específicas para las que fueron diseñados, como por ejemplo predecir el precio de una acción o marcar adecuadamente las transacciones con tarjeta de crédito como fraude.

Debido a que cada modelo de **aprendizaje automático** es único, los métodos óptimos de evaluación varían dependiendo de si el modelo en cuestión es "supervisado" o "no supervisado". Los modelos de aprendizaje automático supervisados hacen predicciones o clasificaciones específicas basadas en datos de entrenamiento etiquetados, mientras que los modelos de aprendizaje automático no supervisados buscan agrupar o encontrar patrones en datos no etiquetados.



Dentro del aprendizaje supervisado se disponen técnicas tanto para tareas de regresión como de clasificación. Si bien algunas técnicas son adecuadas para la regresión o la clasificación, algunas se pueden usar para ambas. Por ejemplo, la regresión lineal solo se puede usar para la regresión, mientras que las máquinas vectoriales de soporte (SVN) y los random forest se pueden usar para cualquiera de los dos. Si bien cada una es una técnica diferente, las métricas que utilizamos para evaluarlos son las mismas, por lo que incluso podemos comparar estos modelos entre sí.

Marcar las compras con **tarjeta de crédito** como fraude es una tarea de clasificación, y predecir los precios de la vivienda, una tarea de regresión.

La tarea de evaluar lo bien que funciona un modelo de aprendizaje supervisado es más sencilla que uno no supervisado. Esto es debido a que los modelos de aprendizaje supervisado aprenden de los datos de entrenamiento etiquetados. Después de que se ha ajustado el modelo utilizando datos de

entrenamiento, se puede probar con datos de la misma población y, por lo tanto, tienen las mismas etiquetas.

Conjuntos de entrenamiento y prueba

Por ejemplo, digamos que necesitamos clasificar si una transacción con tarjeta de crédito es fraudulenta y tenemos un conjunto de datos de transacciones con etiquetas de "fraude" o "no fraude". Podemos entrenar nuestro modelo utilizando todos los datos disponibles, pero esto nos impide evaluarlo de manera adecuada porque no quedan datos "independientes" para las pruebas y el sobreajuste (overfitting) se vuelve difícil de detectar. Este problema se puede evitar dividiendo los datos disponibles en **conjuntos de entrenamiento y prueba**.

El **sobreajuste** ocurre cuando un **modelo** hace **generalizaciones** sobre elementos de datos coincidentes que en realidad no están relacionados con el análisis.

Continuando con el ejemplo de la detección de fraude, el sobreajuste puede ocurrir si el entrenamiento del modelo detecta una correlación entre la longitud del nombre de un cliente (o si el nombre del cliente comienza con una vocal) y la probabilidad de que una transacción sea fraudulenta.

Es probable que las pruebas expongan correlaciones aleatorias y espurias, que después no se repliquen en el conjunto de datos de prueba que se ha mantenido fuera de los datos de entrenamiento. Es probable que un modelo que ha sido "sobreajustado" con sus datos de entrenamiento devuelva una relación de precisión considerablemente menor sobre los datos de prueba.

Esto se puede lograr de varias maneras. Para simplificar, primero hablaremos sobre la división de nuestro conjunto de datos en dos conjuntos:

- Un conjunto de entrenamiento (generalmente el 70% de todo el conjunto de datos) del que aprende el modelo.
- Un conjunto de pruebas (el otro 30%). Debido a que el conjunto de pruebas se retiene del modelo durante el entrenamiento, puede contribuir a una evaluación imparcial de lo bien que funciona un modelo en datos nunca vistos antes. Esto protege contra el sobreajuste y nos permite evaluar cómo funcionaría nuestro modelo en producción, con nuevos datos a medida que surgen.

Validación cruzada

La **validación cruzada** es otro antídoto para el sobreajuste. La validación cruzada implica particionar datos en múltiples grupos y luego entrenar y probar modelos en diferentes combinaciones de grupos.

En una validación cruzada de 5 folds dividiríamos nuestro conjunto de datos de transacciones en cinco particiones de igual tamaño.

Después entrenaríamos nuestro modelo en cuatro de esas cinco particiones y probaríamos nuestro modelo en la partición restante.

Luego repetiríamos el proceso, seleccionando una partición diferente para que sea el grupo de prueba y entrenando un nuevo modelo en el conjunto restante de cuatro particiones.

Repetiríamos tres veces más, para un total de cinco rondas de validación cruzada, una para cada fold.

En este punto tendremos cinco modelos diferentes, cada uno habiendo sido entrenado y probado en un subconjunto diferente de datos y cada uno con sus propios pesos y precisión de predicción.

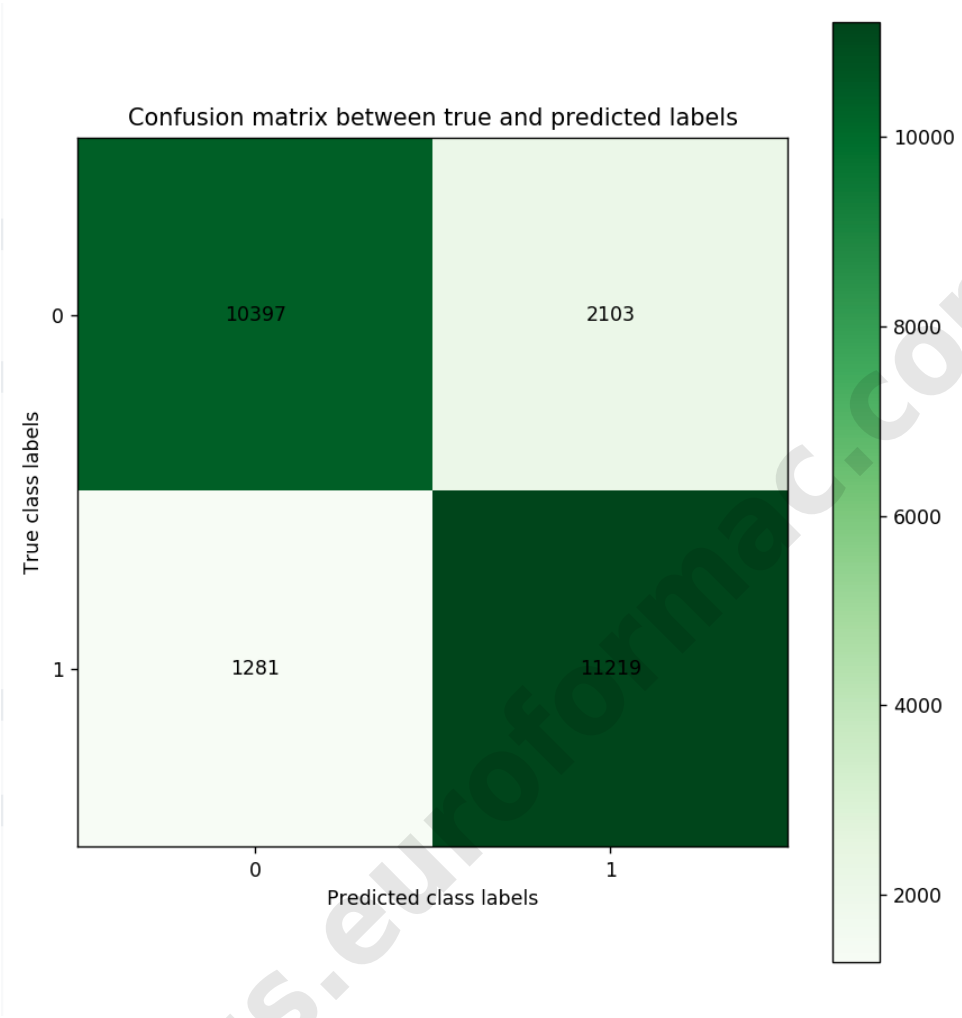
Al final, combinamos estos modelos promediando sus pesos para estimar un modelo predictivo final.

Métricas de clasificación

Las **métricas de clasificación** son las medidas contra las cuales se evalúan los modelos. La métrica más simple y común de este tipo es la **precisión**. La precisión se calcula dividiendo el número de predicciones correctas por el número total de predicciones. En nuestro ejemplo de modelo de clasificación de transacciones supervisadas, si probamos nuestro modelo en cien transacciones y predijimos correctamente su etiqueta (fraude / no fraude) para noventa y cinco de ellas, entonces la precisión de nuestro modelo es del 95%.

La precisión es la métrica más simple y comprensible que podemos usar, pero no podemos confiar solo en la precisión porque no distingue entre falsos positivos, transacciones clasificadas incorrectamente como fraude y falsos negativos, transacciones clasificadas incorrectamente como no fraudulentas. Para ello necesitamos una matriz de confusión.

Una **matriz de confusión** es una tabla de 2 por 2 que clasifica las predicciones en una de cuatro clasificaciones: verdadero positivo, verdadero negativo, falso positivo y falso negativo.



Matriz de confusión

Nuestro modelo de clasificación de transacciones podría generar una matriz de confusión como la siguiente:

La **matriz de confusión** indica que, de 100 transacciones totales, nuestro modelo predijo correctamente el fraude cuatro veces y predijo correctamente no fraude 91 veces, lo que arroja una precisión general del 95%.

La matriz de confusión, sin embargo, también nos permite ver el número de veces que el modelo predijo incorrectamente que una transacción era fraude, un falso positivo que ocurrió en dos de las 100 transacciones. También podemos ver el número de veces que el modelo predijo que una

transacción no era fraude cuando lo era: un falso negativo que ocurrió en tres de las 100 transacciones.

Si bien el modelo parece presumir de una tasa de “verdaderos negativos” bastante buena (el porcentaje de mensajes no fraudulentos clasificados correctamente como tales (91 / (91 + 2) = 97.8%), la tasa de "verdadero positivo" del modelo), el porcentaje de mensajes de fraude correctamente marcados como tales (4 / (4 + 3) = 57.1%) es mucho menos atractiva. Desglosar el rendimiento del modelo de esta manera dibuja una imagen diferente y más completa que la tasa de precisión del 95% por si sola.

Evaluación de modelos de regresión

Los métodos de evaluación también se aplican a los modelos de regresión. Supongamos que tenemos un modelo de regresión que ha sido entrenado para predecir los precios de la vivienda. Los precios pronosticados del modelo se pueden comparar con los precios reales utilizando el **error cuadrático medio**, que mide el promedio de los cuadrados de los errores, que son las diferencias entre el precio real y el previsto. Cuanto menor sea el error cuadrático medio, mejor será el modelo.

Todos los modelos deben ser sometidos a evaluación, cuando se construyen y a lo largo de sus vidas. Los modelos de aprendizaje supervisados y no supervisados plantean diferentes tipos de desafíos de evaluación, y seleccionar el tipo correcto de métricas es clave.

El sobreajuste ocurre cuando un modelo hace generalizaciones sobre elementos de datos coincidentes que en realidad no están relacionados con el análisis.

- Verdadero.
- Falso.

Modelos lineales

Los modelos lineales utilizan una fórmula simple para encontrar la línea de mejor ajuste a través de un conjunto de puntos de datos, y así predecir valores desconocidos.

Los modelos lineales se consideran de la "vieja escuela" y, a menudo, no son tan predictivos como las clases de algoritmos más nuevas, pero se pueden entrenar con relativa rapidez y generalmente son más sencillos de interpretar, lo que puede ser una gran ventaja.

Tenemos dos tipos de modelos lineales:

- Regresión lineal, que se utiliza para la regresión (predicciones numéricas).
- Regresión logística, que se utiliza para la clasificación (predicciones categóricas).

3.1. Regresión lineal simple

La regresión lineal simple es un enfoque para predecir una respuesta utilizando una sola característica.

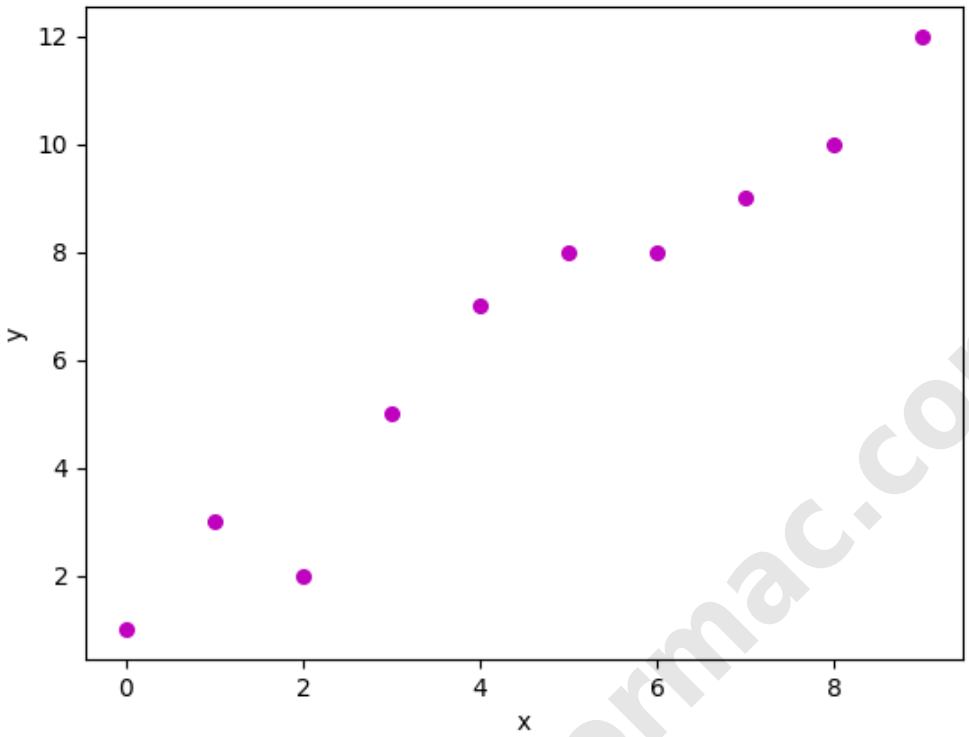
Se supone que las dos variables están relacionadas linealmente. Por lo tanto, tratamos de encontrar una función lineal que prediga el valor de respuesta (y) con la mayor precisión posible como una función de la característica o variable independiente (x).

Consideremos un conjunto de datos donde tenemos un valor de respuesta y para cada característica x:

x	0	1	2	3	4	5	6	7	8	9
y	1	3	2	5	7	8	8	9	10	12

En general, definimos: x como vector de características, es decir, x = [x_1, x_2, …., x_n], y como vector de respuesta, es decir, y= [y_1, y_2, …., y_n] para n observaciones (en el ejemplo anterior, n = 10).

Un gráfico de dispersión del conjunto de datos anterior se ve de la siguiente forma:



Ahora, nuestra tarea es encontrar una línea que se ajuste mejor al diagrama de dispersión anterior para que podamos predecir la respuesta para cualquier nuevo valor de entidad. (es decir, un valor de x no presente en un conjunto de datos)

Esta línea se denomina **línea de regresión**.

La ecuación de la línea de regresión se representa como:

$$h(x_i) = \beta_0 + \beta_1 x_1$$

Donde:

- h(xi) representa el valor de respuesta previsto para la i-ésima observación.
- Beta 0 y beta 1 son coeficientes de regresión y representan la intersección del eje y la pendiente de la línea de regresión respectivamente.

Para crear nuestro modelo, debemos "aprender" o estimar los valores de los coeficientes de regresión beta 0 y beta 1. Y una vez que hemos estimado estos coeficientes, podremos usar el

modelo para predecir respuestas.

3.2 Regresión lineal múltiple

La regresión lineal múltiple intenta modelar la relación entre dos o más características y una respuesta ajustando una ecuación lineal a los datos observados. Lógicamente, no es más que una extensión de la regresión lineal simple.

En este caso consideramos un conjunto de datos con p características (o variables independientes) y una respuesta (o variable dependiente).

Además, el conjunto de datos contiene n filas/observaciones.

3.3 Regresión logística

La clasificación es un área muy importante del aprendizaje automático supervisado. Un gran número de problemas importantes de aprendizaje automático caen dentro de esta área. Hay muchos métodos de clasificación, y la regresión logística es uno de ellos.

Se puede aplicar la clasificación en muchos campos de la ciencia y la tecnología. Por ejemplo, los algoritmos de clasificación de texto se utilizan para separar los correos electrónicos legítimos y no deseados, así como los comentarios positivos y negativos. Otros ejemplos involucran aplicaciones médicas, clasificación biológica, calificación crediticia, etc.

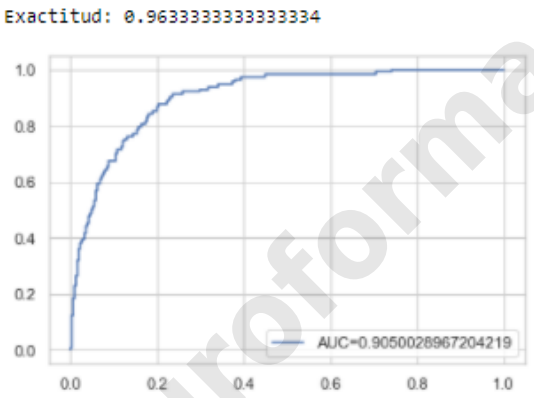
Las tareas de reconocimiento de imágenes a menudo se representan como problemas de clasificación. Por ejemplo, se puede preguntar si una imagen representa un rostro humano o no, o si es un ratón o un elefante, o qué dígito de cero a nueve representa, y así sucesivamente.

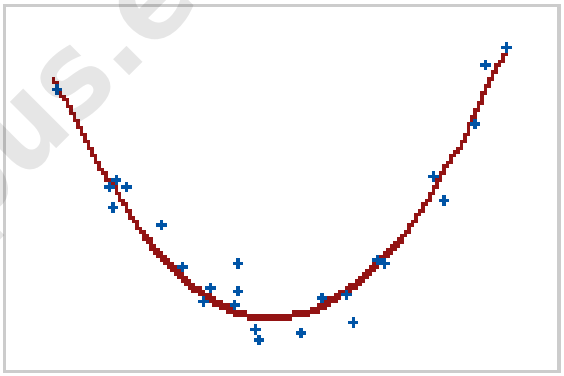
La regresión logística es una técnica de clasificación fundamental. Pertenece al grupo de los clasificadores lineales y es algo similar a la regresión polinómica y lineal. La regresión logística es rápida y relativamente sencilla, y es necesario interpretar los resultados. Aunque es esencialmente un método para la clasificación binaria, también se puede aplicar a problemas multiclase.

3.4 Suposiciones sobre el conjunto de datos

A continuación, revisamos los supuestos básicos que un modelo de regresión lineal hace con respecto a un conjunto de datos en el que se aplica:

Relación lineal: La relación entre las variables de respuesta y de característica debe ser lineal. La suposición de linealidad se puede probar utilizando diagramas de dispersión. Como se muestra a continuación, la 1ª figura representa variables relacionadas linealmente, mientras que las variables de la 2ª figura son probablemente no lineales. Por lo tanto, la 1ª figura dará mejores predicciones utilizando la regresión lineal.

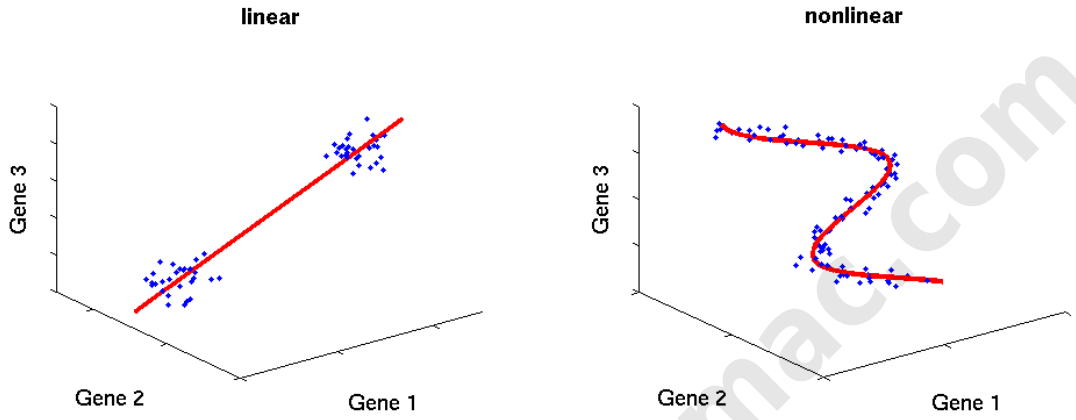




Poca o ninguna multicolinealidad: Se supone que hay poca o ninguna multicolinealidad en los datos. La multicolinealidad ocurre cuando las características (o variables independientes) no son independientes entre sí.

Poca o ninguna autocorrelación: Otra suposición es que hay poca o ninguna autocorrelación en los datos. La autocorrelación ocurre cuando los errores residuales no son independientes entre sí.

Homocedasticidad: La homocedasticidad describe una situación en la que el término error (es decir, el "ruido" o perturbación aleatoria en la relación entre las variables independientes y la variable dependiente) es el mismo en todos los valores de las variables independientes. Como se muestra a continuación, la figura 1 tiene homocedasticidad, mientras que la figura 2 tiene heterocedasticidad.



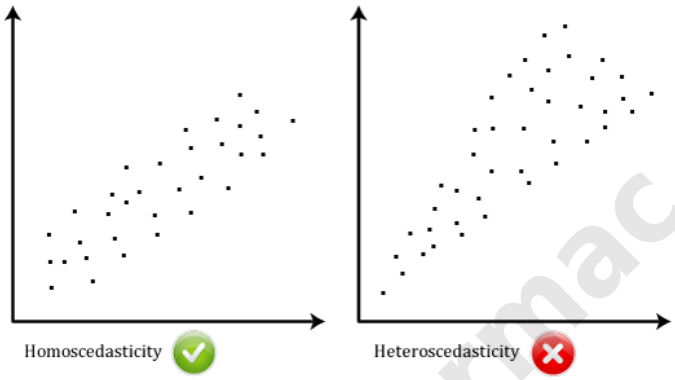
La regresión lineal simple es un enfoque para predecir una respuesta utilizando una sola característica.

- Verdadero.
- Falso.

Modelos supervisados de ML: árboles, SVM, redes neuronales.

Árboles de decisión

Un árbol de decisión es una herramienta de apoyo a la decisión que utiliza un gráfico o modelo de decisiones en forma de árbol y sus posibles consecuencias, incluidos los resultados de eventos fortuitos, los costes de recursos y la utilidad.



Desde el punto de vista de la decisión comercial, un árbol de decisión es el número mínimo de preguntas de sí / no que uno tiene que hacer, para evaluar la probabilidad de tomar una decisión correcta (en la mayoría de los casos). Como método, nos permite abordar el problema de una manera estructurada y sistemática para llegar a una conclusión lógica.

El **árbol de decisión** divide la muestra en dos o más conjuntos homogéneos (hojas) en función de los diferenciadores más significativos en sus variables de entrada. Para elegir un diferenciador (predictor), el algoritmo considera todas las características y realiza una división binaria en ellas (para datos categóricos, divididos por cat; para continuo, se elige un umbral de corte). Luego elegirá el que tenga el menor coste (es decir, la mayor precisión) y se repita recursivamente, hasta que divida con éxito los datos en todas las hojas (o alcance la profundidad máxima).

La principal ventaja del árbol de decisión es que es fácil de entender y visualizar, requiere poca preparación de datos y puede manejar datos numéricos y categóricos.

Como desventaja, el árbol de decisión puede crear árboles complejos que no se generalizan bien, y los árboles de decisión pueden ser inestables porque pequeñas variaciones en los datos pueden generar un árbol completamente diferente.

Implementación de un árbol de decisión

Usaremos la biblioteca scikit-learn para construir el modelo y usar el conjunto de datos de iris que ya está incluido en la biblioteca scikit-learn.

El conjunto de datos contiene tres clases de plantas: Iris Setosa, Iris Versicolour, Iris Virginica con los siguientes atributos:

- Longitud del sépalo.
- Ancho del sépalo.
- Longitud de pétalos.
- Ancho de pétalos.

Tenemos que predecir la clase de la planta de iris en función de sus atributos.

Como siempre, importamos las bibliotecas requeridas.

import pandas as pd

import numpy as np

from sklearn.datasets import load_iris

from sklearn import tree

Cargamos el conjunto de datos de iris

iris=load_iris()

Para ver todas las características del datset, utilizamos la función de impresión

print(iris.feature_names)

Salida: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

Para ver todos los nombres de destino en el conjunto de datos:

print(iris.target_names)

Salida: ['setosa' 'versicolor' 'virginica']

Para tener datos de **verificación**, vamos a quitar unas etiquetas, para ello eliminamos los elementos en la 0ª, 50ª y 100ª posición.

- El 0º elemento pertenece a la especie Setosa.
- El 50º pertenece a la especie Versicolor.
- El 100º pertenece a la especie Virginica.

Esto eliminará las etiquetas para que podamos entrenar mejor a nuestro clasificador de **árbol de decisión** y verificar si es capaz de clasificar bien los datos (utilizando los datos eliminados).

```
removed =[0,50,100]

new_target = np.delete(iris.target,removed)

new_data = np.delete(iris.data,removed, axis=0)
```

Para entrenar el árbol de decisión, utilizamos un clasificador de árbol de decisión de scikit-learn para la clasificación.

```
#Entrenar clasificador

clf = tree.DecisionTreeClassifier() # definir árbol de decisión clasificador

clf=clf.fit(new_data,new_target) # entrenar datos sobre new_data y new_target

prediction = clf.predict(iris.data[removed]) # asignar los datos eliminados a la entrada
```

Ha llegado el momento de comprobamos si nuestras etiquetas predichas coinciden con las etiquetas originales.

```
print("Original Labels",iris.target[removed])

print("Labels Predicted",prediction)
```

Salida:

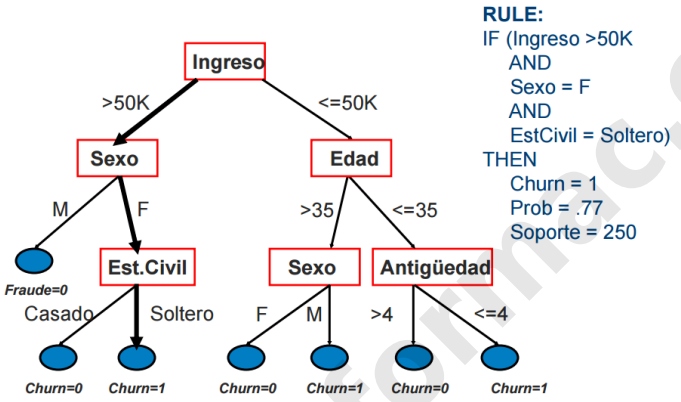
Original Labels [0 1 2]

Labels Predicted [0 1 2]

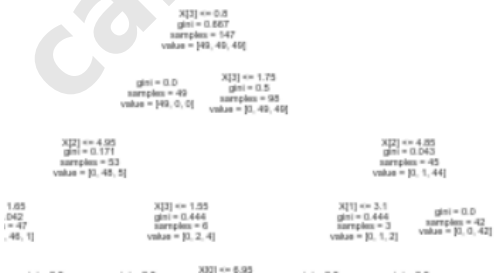
La precisión de nuestro modelo es del 100%.

Finalmente, dibujamos nuestro árbol de decisión:

tree.plot_tree(clf)




```
.50769230769231, 90.6, 'X[3] <= 1.65\ngini = 0.042\nsamples = 47\nvalue = [0, 46, 1
.753846153846155, 54.359999999999985, 'gini = 0.0\nsamples = 46\nvalue = [0, 46, 0]
.26153846153846, 54.359999999999985, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
4.52307692307693, 90.6, 'X[3] <= 1.55\ngini = 0.444\nsamples = 6\nvalue = [0, 2, 4]
8.76923076923077, 54.359999999999985, 'gini = 0.0\nsamples = 3\nvalue = [0, 0, 3]')
0.27692307692308, 54.359999999999985, 'X[0] <= 6.95\ngini = 0.444\nsamples = 3\nval
4.52307692307693, 18.119999999999976, 'gini = 0.0\nsamples = 2\nvalue = [0, 2, 0]')
6.03076923076924, 18.119999999999976, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]')
3.2923076923077, 126.83999999999999, 'X[2] <= 4.85\ngini = 0.043\nsamples = 45\nval
7.53846153846155, 90.6, 'X[1] <= 3.1\ngini = 0.444\nsamples = 3\nvalue = [0, 1, 2]'
1.7846153846154, 54.359999999999985, 'gini = 0.0\nsamples = 2\nvalue = [0, 0, 2]'),
3.2923076923077, 54.359999999999985, 'gini = 0.0\nsamples = 1\nvalue = [0, 1, 0]'),
9.04615384615386, 90.6, 'gini = 0.0\nsamples = 42\nvalue = [0, 0, 42]')]
```



Fuente: https://scikit-learn.org/stable/modules/tree.html

A continuación, se lista el **código completo**:

```
#IMPORTAR PAQUETES

import pandas as pd

import numpy as np

from sklearn.datasets import load_iris

from sklearn import tree

iris=load_iris()

print(iris.feature_names)

print(iris.target_names)

#Quitar etiquetas de los datos

removed =[0,50,100]

new_target = np.delete(iris.target,removed)

new_data = np.delete(iris.data,removed, axis=0)

#Entrenar clasificador

clf = tree.DecisionTreeClassifier() # definir árbol de decisión clasificador

clf=clf.fit(new_data,new_target) # entrenar datos sobre new_data y new_target

prediction = clf.predict(iris.data[removed]) # asignar los datos eliminados a la entrada

#Verificar con etiquetas borradas

print("Original Labels",iris.target[removed])

print("Labels Predicted",prediction)

#Dibujar árbol

tree.plot_tree(clf)
```

SVM Máquinas de vectores soporte (Support Vector Machine)

Las **máquinas vectoriales de soporte (SVM)** son algoritmos de aprendizaje automático supervisado potentes pero flexibles que se utilizan tanto para la clasificación como para la regresión. Pero generalmente, se utilizan en problemas de clasificación.

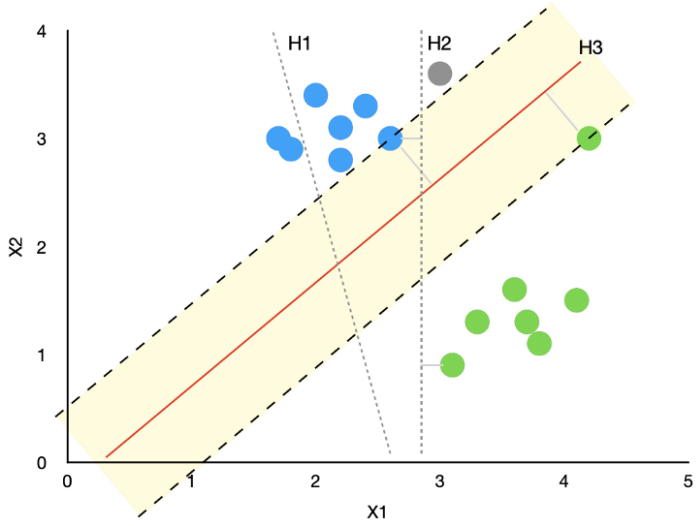
En la década de 1960, los SVM se introdujeron por primera vez, y se refinaron en 1990. Los SVM tienen una forma única de implementación en comparación con otros algoritmos de aprendizaje automático. En la actualidad, son muy populares debido a su capacidad para trabajar con múltiples variables continuas y categóricas.

La máquina de vectores de soporte es una representación de los datos de entrenamiento como puntos en el espacio separados en categorías por una brecha clara que es lo más amplia posible. Luego se asignan nuevos ejemplos en ese mismo espacio y se predice que pertenecen a una categoría en función de qué lado de la brecha caen.

La **ventaja de SVM** es que es efectivo en espacios de altas dimensiones y utiliza un subconjunto de puntos de entrenamiento en la función de decisión, por lo que también es eficiente en la memoria.

Como desventaja, el algoritmo no proporciona directamente estimaciones de probabilidad, estas estimaciones se calculan utilizando una costosa validación cruzada de cinco folds.

Un modelo SVM es básicamente una representación de diferentes clases en un hiperplano en el espacio multidimensional. El hiperplano será generado de manera iterativa por SVM para que el error pueda ser minimizado. El objetivo de SVM es dividir los conjuntos de datos en clases para encontrar un hiperplano marginal máximo (MMH).



En la imagen sobre hiperplanos, vemos lo siguiente:

- H1 no es un buen hiperplano, ya que no separa las clases.
- H2 separa, pero solo con un margen pequeño.
- H3 los separa con el margen máximo (distancia).

Parámetros de SVM

- Vectores de soporte: los puntos de datos que están más cerca del hiperplano se denominan vectores de soporte. La línea de separación se definirá con la ayuda de estos puntos de datos.
- Hiperplano: es un plano de decisión o espacio que se divide entre un conjunto de objetos que tienen diferentes clases.
- Margen: se puede definir como la brecha entre dos líneas en los puntos de datos más cercanos clases diferentes. Se puede calcular como la distancia perpendicular desde la línea hasta los vectores de soporte. El margen grande se considera como un margen bueno y el margen pequeño se considera como un margen malo.

El objetivo principal de SVM es dividir los conjuntos de datos en clases para encontrar un hiperplano marginal máximo (MMH). Esto se puede hacer en los siguientes dos pasos:

- SVM generará hiperplanos iterativamente que segregan las clases de la mejor manera.
- SVM elegirá el hiperplano que separa las clases correctamente.

Implementación de SVM

Para implementar SVM en Python comenzaremos con la importación de bibliotecas estándar de la siguiente manera:

import numpy as np

import matplotlib.pyplot as plt

from scipy import stats

import seaborn as sns; sns.set()

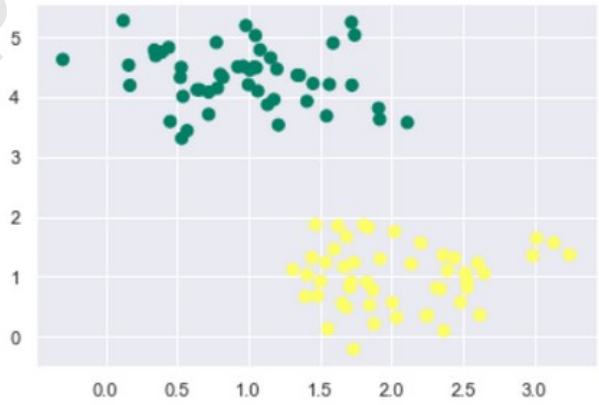
A continuación, creamos un conjunto de datos de muestra, que tiene datos linealmente separables, a partir de sklearn.dataset.sample_generator para su clasificación utilizando SVM:

" from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=100, centers=2, random_state=0, cluster_std=0.50)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer'); "

Vemos cuál sería el resultado después de generar un conjunto de datos de muestra que tenga 100 muestras y 2 clústeres:



El árbol de decisión divide la muestra en dos o más conjuntos homogéneos (hojas) en función de los diferenciadores más significativos en sus variables de

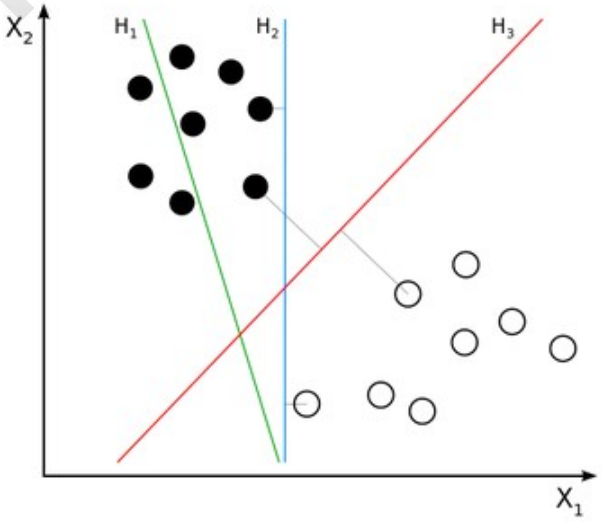
entrada. Para elegir un diferenciador (predictor), el algoritmo considera todas las características y realiza una división binaria en ellas (para datos categóricos, divididos por cat; para continuo, se elige un umbral de corte).

Verdadero.
Falso.

Sabemos que **SVM** soporta la clasificación discriminativa. divide las **clases** entre sí simplemente encontrando una línea en caso de dos dimensiones o una variedad en caso de múltiples dimensiones. Para implementarla en el conjunto de datos anterior lo haremos de la siguiente manera:

```
xfit = np.linspace(-1, 3.5)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

plt.plot([0.6], [2.1], 'x', color='black', markeredgewidth=4, markersize=12)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:

plt.plot(xfit, m * xfit + b, '-k')

plt.xlim(-1, 3.5);
```

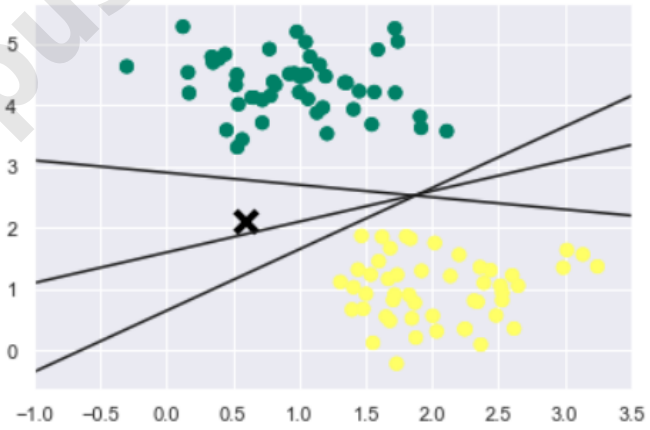
El resultado es el siguiente:



Podemos ver en la salida anterior que hay tres separadores diferentes que discriminan perfectamente las muestras anteriores.

Como hemos visto, el objetivo principal de SVM es dividir los conjuntos de datos en clases para encontrar un hiperplano marginal máximo (MMH), por lo tanto, en lugar de dibujar una línea cero entre las clases, podemos dibujar alrededor de cada línea un margen de cierto ancho hasta el punto más cercano. Se puede hacer de la siguiente forma:

```
xfit = np.linspace(-1, 3.5)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:

yfit = m * xfit + b

plt.plot(xfit, yfit, '-k')

plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',

color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);
```



A partir de la **imagen de arriba** en la salida, podemos observar fácilmente los "márgenes" dentro de los clasificadores discriminativos. SVM elegirá la línea que maximice el margen.

A continuación, utilizaremos el clasificador de vectores de soporte (SVC) de Scikit-Learn para

entrenar un modelo SVM sobre estos datos. Aquí, utilizaremos el kernel lineal para ajustarse a SVM de la siguiente forma:

```
from sklearn.svm import SVC # "Support vector classifier"

model = SVC(kernel='linear', C=1E10)

model.fit(X, y)
```

El resultado es el siguiente:

```
SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0,

decision_function_shape='ovr', degree=3, gamma='auto_deprecated',

kernel='linear', max_iter=-1, probability=False, random_state=None,

shrinking=True, tol=0.001, verbose=False)
```

Ahora, para entender mejor el proceso, definimos una función de decisión para 2D SVC:

```
def decision_function(model, ax=None, plot_support=True):

if ax is None:

ax = plt.gca()

xlim = ax.get_xlim()

ylim = ax.get_ylim()
```

Para evaluar el modelo, necesitamos crear una cuadrícula de la siguiente forma:

```
x = np.linspace(xlim[0], xlim[1], 30)

y = np.linspace(ylim[0], ylim[1], 30)

Y, X = np.meshgrid(y, x)

xy = np.vstack([X.ravel(), Y.ravel()]).T
```

P = model.decision_function(xy).reshape(X.shape)

A continuación, debemos trazar los límites y márgenes de decisión de la siguiente manera:

ax.contour(X, Y, P, colors='k',

levels=[-1, 0, 1], alpha=0.5,

linestyles=['--', '-', '--'])

Ahora, de manera similar, trazamos los **vectores de soporte** de la siguiente manera:

if plot_support:

ax.scatter(model.support_vectors_[:, 0],

model.support_vectors_[:, 1],

s=300, linewidth=1, facecolors='none');

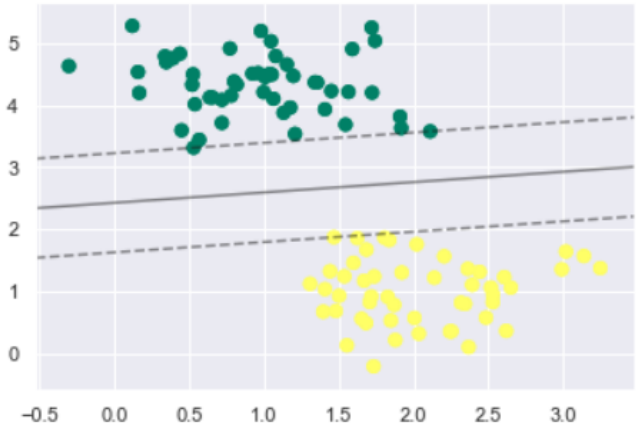
ax.set_xlim(xlim)

ax.set_ylim(ylim)

Ahora, usamos la función para adaptarse a nuestro modelo de la siguiente manera:

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

decision_function(model);



Podemos observar en la salida anterior que un **clasificador SVM** se ajusta a los datos con márgenes, es decir, líneas discontinuas y vectores de soporte, los elementos fundamentales de este ajuste, tocando la línea discontinua. Estos puntos vectoriales de soporte se almacenan en el atributo **support_vectors_** del clasificador de la siguiente manera:

```
model.support_vectors_
```

El resultado es el siguiente:

```
array([[0.5323772 , 3.31338909],
[2.11114739, 3.57660449],
[1.46870582, 1.86947425]])
```

El código completo es el siguiente:

```
#IMPORTAR PAQUETES
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns; sns.set()
#Crear dataset
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=100, centers=2, random_state=0, cluster_std=0.50)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer');
#Incluir separadores
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')
```

```
plt.plot([0.6], [2.1], 'x', color='black', markeredgewidth=4, markersize=12)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:

plt.plot(xfit, m * xfit + b, '-k')

plt.xlim(-1, 3.5);

#Dar margen a los separadores

xfit = np.linspace(-1, 3.5)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:

yfit = m * xfit + b

plt.plot(xfit, yfit, '-k')

plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none', color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);

#Utilizar SVC

from sklearn.svm import SVC # "Support vector classifier"

model = SVC(kernel='linear', C=1E10)

model.fit(X, y)

#2D SVC

def decision_function(model, ax=None, plot_support=True):

if ax is None:

ax = plt.gca()

xlim = ax.get_xlim()
```

```
ylim = ax.get_ylim()

#Crear cuadrícula

x = np.linspace(xlim[0], xlim[1], 30)

y = np.linspace(ylim[0], ylim[1], 30)

Y, X = np.meshgrid(y, x)

xy = np.vstack([X.ravel(), Y.ravel()]).T

P = model.decision_function(xy).reshape(X.shape)

#Dibujar límites y márgenes

ax.contour(X, Y, P, colors='k',

levels=[-1, 0, 1], alpha=0.5,

linestyles=['--', '-', '--'])

#Dibujar SVC

if plot_support:

ax.scatter(model.support_vectors_[:, 0],

model.support_vectors_[:, 1],

s=300, linewidth=1, facecolors='none');

ax.set_xlim(xlim)

ax.set_ylim(ylim)

#Dibujar resultado

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

decision_function(model);
```

#Mostrar vectores de soporte

model.support_vectors_

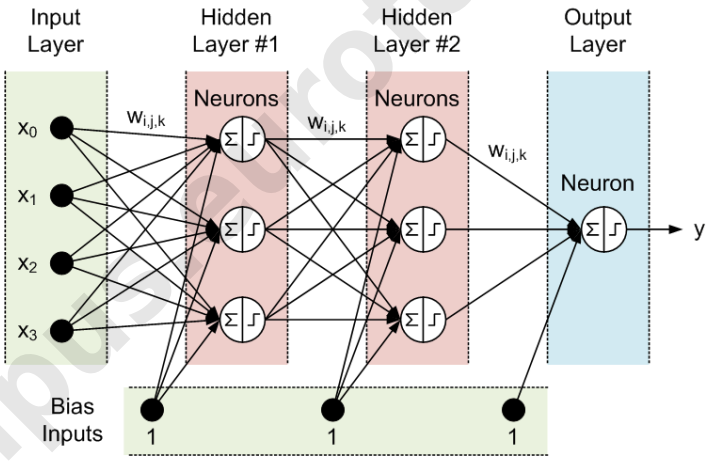
Redes neuronales

La red neuronal artificial, o simplemente red neuronal para abreviar, no es una idea nueva, existe desde hace más de 80 años.

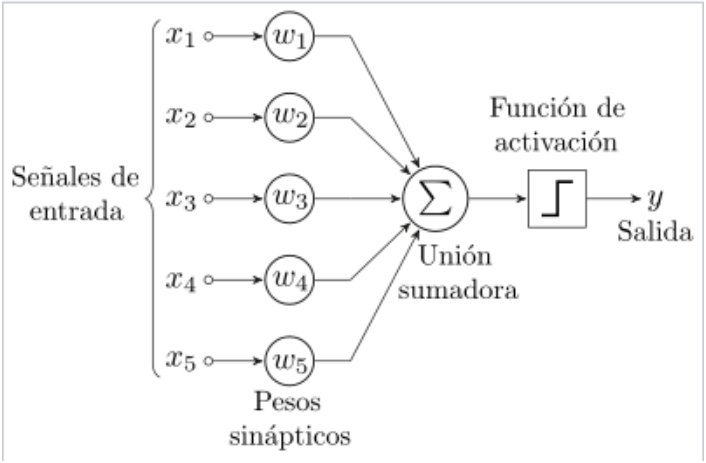
No fue hasta 2011, cuando **Deep Neural Networks** se hizo popular gracias al desarrollo de nuevas técnicas, gran disponibilidad de conjuntos de datos y computadoras potentes.

Como ya hemos visto, una red neuronal imita una neurona, que tiene dendritas, un núcleo, un axón y un axón terminal. Para una red, necesitamos dos neuronas. Estas neuronas transfieren información a través de la sinapsis entre las dendritas de uno y el axón terminal de otro.

Un modelo probable de una neurona artificial será así:



Una red neuronal tendrá el aspecto siguiente:



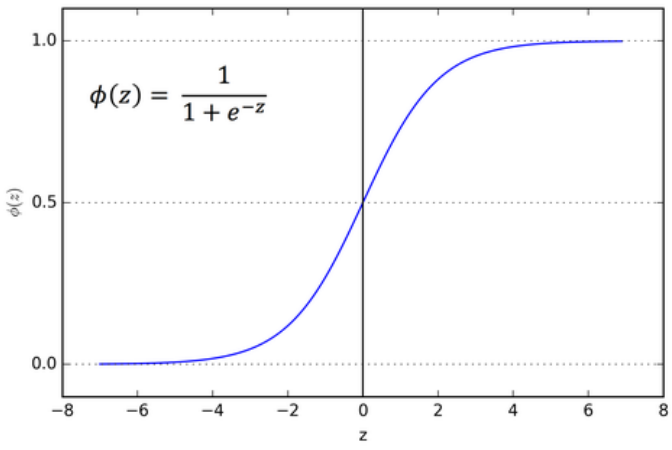
Los círculos son **neuronas o nodos**, con sus funciones en los datos. Las líneas que los conectan son los pesos o la información que se transmiten.

Cada columna es una capa. La primera capa de los datos es la capa de entrada. Todas las capas entre la capa de entrada y la capa de salida son las capas ocultas.

Si tenemos una o algunas capas ocultas, entonces tenemos una red neuronal poco profunda. Si tenemos muchas capas ocultas, entonces estamos ante una red neuronal profunda.

En este modelo, se parte de unos datos de entrada, los pesa y los pasa a través de la función en la neurona que se llama función de umbral o función de activación. Básicamente, es la suma de todos los valores después de compararlo con un cierto valor. Si dispara una señal, entonces el resultado de salida es (1), o no se dispara nada, entonces se retorna (0). Luego se pondera y pasa a la siguiente neurona, y se ejecuta el mismo tipo de función.

Podemos tener una función **sigmoidea (forma de s)** como la función de activación.



En cuanto a los pesos, son aleatorios para comenzar, y son únicos por entrada en el nodo/neurona.

El acto de enviar datos directamente a través de una red neuronal se denomina red neuronal de avance (**feed forward**).

En un "feed forward" típico, el tipo más básico de red neuronal, la información pasa directamente a través de la red creada, y se compara la salida con lo que se esperaba que la salida fuera utilizando los datos de muestra.

A partir de aquí, deberemos ajustar los pesos para conseguir que la salida coincida con la salida deseada.

Nuestros datos van desde la entrada, a las capas, en orden, luego a la salida.

Cuando retrocedemos y comenzamos a ajustar los pesos para minimizar la pérdida / coste esto se denomina propagación hacia atrás (**back propagation**).

Estamos ante un problema de optimización. En la práctica real, es habitual tener que lidiar con cientos de miles de variables, o millones, o incluso más.

La primera solución fue utilizar el **descenso de gradiente estocástico** como método de optimización. Ahora, hay opciones como **AdaGrad**, **Adam Optimizer,** etc. De cualquier manera, esta es una operación computacional masiva. Esta es la razón por la que las redes neuronales no avanzaron en más de medio siglo. Solamente en tiempos recientes se ha dispuesto de máquinas suficientemente potentes para trabajar con estos datos.

Para tareas de clasificación simples, la red neuronal tiene un rendimiento relativamente cercano a otros algoritmos simples como el vecino más cercano (K Nearest Neighbors). La utilidad real de las redes neuronales se manifiesta cuando tenemos datos mucho más grandes y preguntas mucho más complejas, que superan a otros modelos de aprendizaje automático.

Los círculos son neuronas o nodos, con sus funciones en los datos. Las líneas que los conectan son los pesos o la información que se transmiten.

Verdadero.

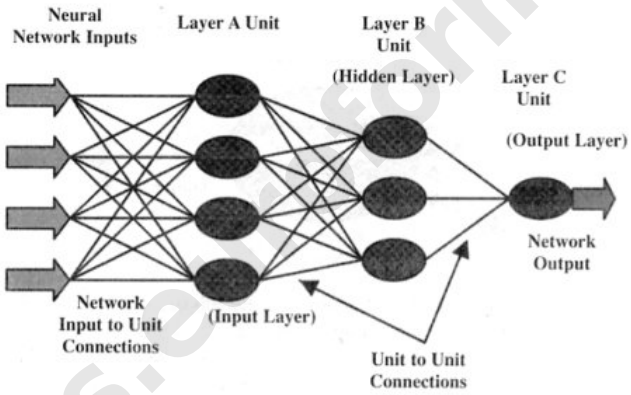
Falso.

Redes neuronales profundas

Una red neuronal profunda (DNN) es una red neuronal artificial (ANN) con múltiples capas ocultas entre las capas de entrada y salida. Al igual que los ANN poco profundos, los DNN pueden modelar relaciones complejas no lineales.

El objetivo principal de una red neuronal es recibir un conjunto de entradas, realizar cálculos progresivamente complejos en ellas y dar salida para resolver problemas del mundo real como la clasificación.

Figure 3: A 3-layer feed-forward neural networks



Nosotros nos limitaremos a alimentar las redes neuronales. En una red profunda, tenemos una entrada, una salida y un flujo de datos secuenciales.

Las redes neuronales se usan ampliamente en el aprendizaje supervisado y los problemas de aprendizaje de refuerzo. Estas redes se basan en un conjunto de capas conectadas entre sí.

En el aprendizaje profundo, el número de capas ocultas, en su mayoría no lineales, puede ser grande, podemos hablar de unas 1000 capas.

Los modelos DL (Deep Learning) producen resultados mucho mejores que las redes ML (Machine Learning) normales.

Utilizamos principalmente el **método de descenso** de gradiente para optimizar la red y minimizar

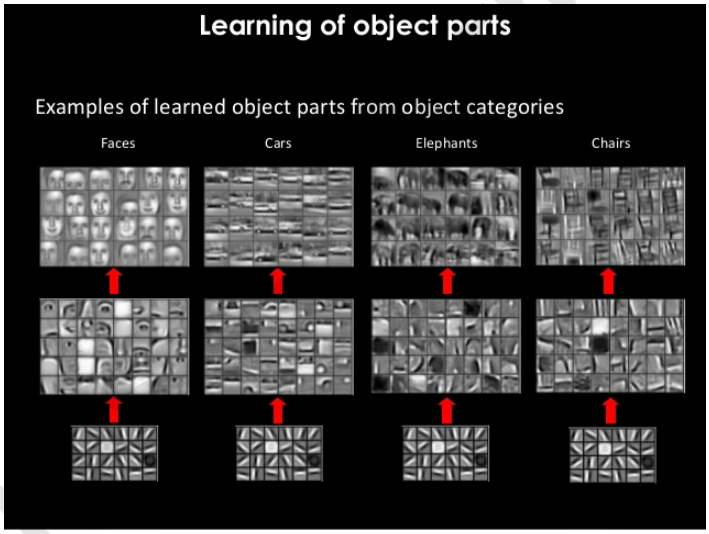
pérdida de la función.

Podemos usar **Imagenet**, un repositorio de millones de imágenes digitales para clasificar un conjunto de datos en categorías como gatos y perros. Las redes DL se utilizan cada vez más sobre imágenes dinámicas aparte de las estáticas y para series de tiempo y análisis de texto.

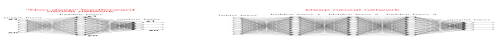
El entrenamiento de los conjuntos de datos forma una parte importante de los modelos de Deep Learning. La retropropagación es el algoritmo principal en el entrenamiento de modelos DL.

DL se ocupa de entrenar grandes redes neuronales con transformaciones complejas de entrada y salida.

Un ejemplo de DL es el mapeo de una foto con el nombre de la (s) persona (s) en la foto como lo hacen en las redes sociales y describir una imagen con una frase es otra aplicación reciente de DL.



El cálculo de los valores de cada nodo se calcularía de la siguiente forma:



$$y_1 = \sigma[(w]_1, 1\, x_1 + w_1, 2\, x_2 + w_1, 3\, w_3)$$

Las redes neuronales son funciones que tienen entradas como x1, x2, x3 … xn, que se transforman en salidas como z1, z2, z3, … zn y así sucesivamente en dos (redes superficiales) o en varias operaciones intermedias también llamadas capas (redes profundas).

Los pesos y sesgos cambian de capa a capa. 'w' y 'v' son los pesos o sinapsis de las capas de las redes neuronales.

El mejor caso de uso del aprendizaje profundo es el problema de **aprendizaje supervisado**. Aquí, tenemos un gran conjunto de entradas de datos con un conjunto deseado de salidas.

$$z_1 = \sigma[(v]_1, 1\, y_1 + v_1, 3\, x_3 + w_1, 3\, y_3 + w_1, 4\, y_4)$$

Aquí aplicamos el algoritmo de propagación hacia atrás para obtener una predicción de salida correcta.

El conjunto de datos más básico del aprendizaje profundo es el MNIST, un conjunto de datos de dígitos escritos a mano. Es posible entrenar en profundidad una red neuronal convolucional con Keras para clasificar imágenes de dígitos escritos a mano de este conjunto de datos.

El disparo o la activación de un clasificador de redes neuronales produce una puntuación. Por ejemplo, para clasificar a los pacientes como enfermos y sanos, consideramos parámetros como la altura, el peso y la temperatura corporal, la presión arterial, etc. Un puntaje alto significa que el paciente está enfermo y un puntaje bajo significa que está sano.

Cada nodo en la salida y las capas ocultas tiene sus propios clasificadores. La capa de entrada toma entradas y pasa sus puntajes a la siguiente capa oculta para una mayor activación y esto continúa hasta que se alcanza la salida.

Este progreso de entrada a salida de izquierda a derecha en la dirección hacia adelante se llama **propagación hacia adelante (forward propagation)**.

La **ruta de asignación de crédito (Credit Assignment Path - CAP)** en una red neuronal es la serie de transformaciones que comienzan desde la entrada hasta la salida. Los CAP elaboran conexiones causales probables entre la entrada y la salida.

La profundidad CAP para una red neuronal de propagación hacia delante dada o la profundidad CAP es el número de capas ocultas más una. Porque también se incluye la capa de salida. Para redes neuronales recurrentes, donde una señal puede propagarse a través de una capa varias veces, la profundidad CAP puede ser potencialmente ilimitada.

Redes profundas y redes poco profundas

No hay un umbral claro de profundidad que divida el aprendizaje superficial del aprendizaje profundo. Pero se considera que para el aprendizaje profundo que tiene múltiples capas no lineales, el CAP (Credit Assignment Path) debe ser mayor que dos.

El nodo básico en una red neuronal es un perceptrón que imita una neurona en una red neuronal biológica. Luego tenemos Percepción multicapa o MLP. Cada conjunto de entradas es modificado por un conjunto de pesos y sesgos. Cada borde tiene un peso único y cada nodo tiene un sesgo único.

La precisión de predicción de una red neuronal depende de sus pesos y sesgos.

El proceso de mejorar la precisión de la red neuronal se llama **entrenamiento**. La salida de una red de propagación hacia delante se compara con el valor que se sabe que es correcto.

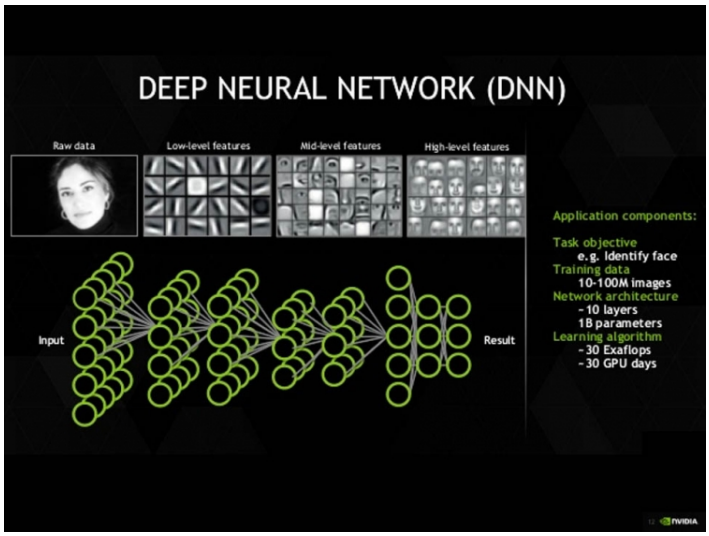
La **función de coste o la función de pérdida** es la diferencia entre el producto generado y el producto real.

El objetivo del entrenamiento es hacer que el coste del entrenamiento sea lo más pequeño posible en millones de ejemplos de entrenamiento. Para hacer esto, la red ajusta los pesos y los sesgos hasta que la predicción coincida con la salida correcta.

Una vez que se entrena bien, una red neuronal tiene el potencial de hacer una predicción precisa cada vez.

Cuando el patrón se vuelve complejo y necesitamos que el ordenador los reconozca, debemos buscar redes neuronales. En estos escenarios de patrones complejos, la red neuronal supera a todos los demás algoritmos competidores.

Ahora hay GPU (Graphic Proceessing Unit), que pueden entrenarlas más rápido que nunca. Las redes neuronales profundas ya están revolucionando el campo de la IA.



Las computadoras han demostrado ser buenas para realizar cálculos repetitivos y seguir instrucciones detalladas, pero hasta ahora no han sido tan buenas para reconocer patrones complejos.

Para tratar el problema del **reconocimiento de patrones simples**, podemos utilizar una máquina de vectores de soporte (support vector machine, svm) o un clasificador de regresión logística, pero a medida que la complejidad del patrón aumenta, no hay más remedio que buscar redes neuronales profundas.

Por lo tanto, para patrones complejos como un rostro humano, las redes neuronales superficiales fallan y no tienen otra alternativa que buscar redes neuronales profundas con más capas. Las redes profundas pueden hacer este trabajo al descomponer los patrones complejos en otros más simples. Por ejemplo, rostro humano; una red profunda usaría bordes para detectar partes como labios, nariz, ojos, oídos, etc., y luego volvería a combinarlos para formar un rostro humano.

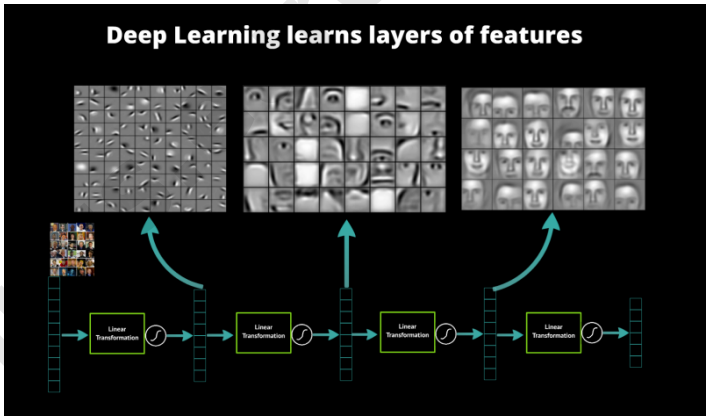
La precisión de la predicción correcta se ha vuelto tan precisa que recientemente, en un **Google Pattern Recognition Challenge**, una red profunda venció a un humano.

Esta idea de una red de perceptrones en capas ha existido desde hace tiempo. En esta área, las redes profundas imitan el cerebro humano. Pero una desventaja de esto es que tardan mucho tiempo en entrenarse, una restricción impuesta por el hardware.

Sin embargo, las GPU recientes de alto rendimiento han podido entrenar redes tan profundas en menos de una semana. Mientras que las CPUs más rápidas podrían haber tardado semanas o quizás meses para hacer lo mismo.

Redes de una sola capa

Como ya sabemos, las redes neuronales artificiales (ANN) son el sistema de procesamiento de información cuyo mecanismo está inspirado en la funcionalidad de los circuitos neuronales biológicos. Una red neuronal artificial se compone de muchas unidades de procesamiento conectadas entre sí.



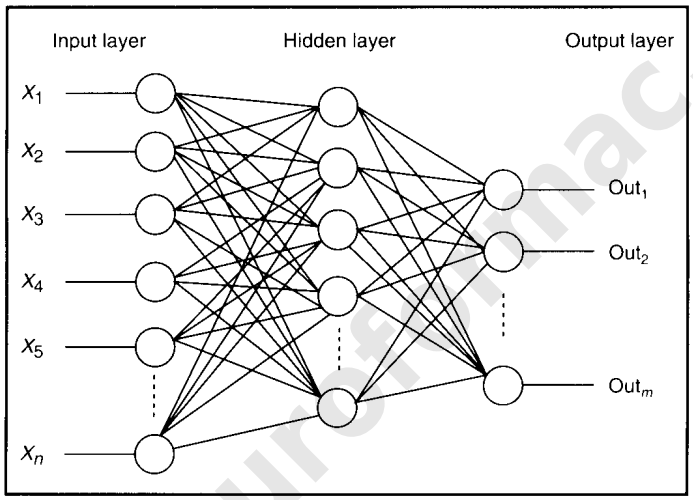
El diagrama muestra que las unidades ocultas se comunican con la capa externa. Mientras que las unidades de entrada y salida se comunican solo a través de la capa oculta de la red.

El patrón de conexión con los nodos, el número total de capas y el nivel de nodos entre las entradas y las salidas con el número de neuronas por capa definen la arquitectura de una red neuronal.

Hay dos tipos de arquitectura. Estos tipos se centran en la funcionalidad de las redes neuronales artificiales de la siguiente manera:

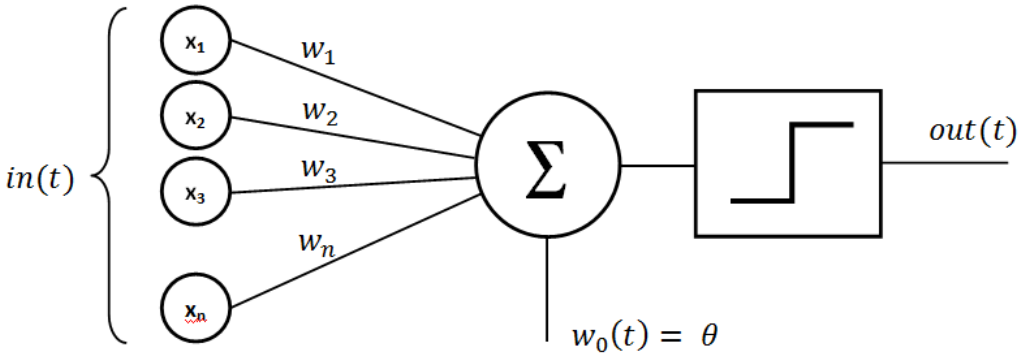
- Perceptrón de una capa.
- Perceptrón multicapa.

El **perceptrón de capa única** es el primer modelo neural propuesto creado. El contenido de la memoria local de la neurona consiste en un vector de pesos. El cálculo de un perceptrón de capa única se realiza sobre el cálculo de la suma del vector de entrada, cada uno con el valor multiplicado por el elemento correspondiente del vector de los pesos. El valor que se muestra en la salida será la entrada de una función de activación.



El algoritmo perceptrón fue diseñado para clasificar entradas visuales, categorizar sujetos en uno de dos tipos y separar grupos con una línea. La clasificación es una parte importante del aprendizaje automático y el procesamiento de imágenes. Los algoritmos de aprendizaje automático encuentran y clasifican patrones por muchos medios diferentes. El algoritmo perceptrón clasifica patrones y grupos al encontrar la separación lineal entre diferentes objetos y patrones que se reciben a través de entradas numéricas o visuales.

Por otro lado, el **perceptrón multicapa** es una **red neuronal artificial** formada por múltiples capas de perceptrones, esto le permite resolver problemas que no son linealmente separables, lo cual es la principal limitación del perceptrón. El perceptrón multicapa es un subconjunto de las redes neuronales profundas (DNN). Si bien DNN puede tener bucles y el perceptrón multicapa siempre es feed forward (pase hacia adelante).

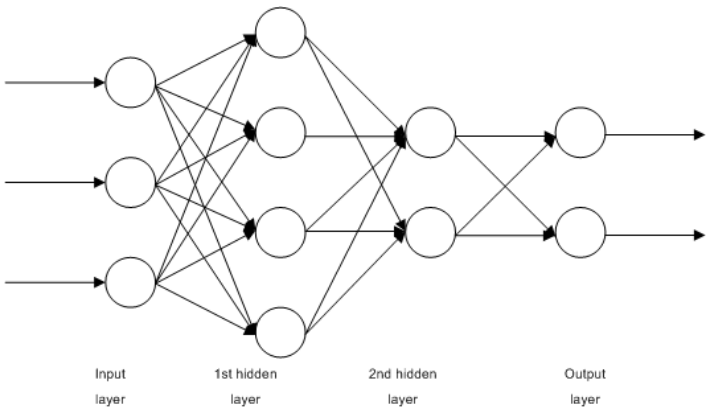


La distribución de capas es la siguiente:

- **Capa de entrada:** Las capas de entrada proporcionan información del mundo exterior (entorno) a la red.
- **Capas ocultas:** Las capas ocultas realizan cálculos y transfieren información desde la capa de entrada a las capas de salida. Las capas ocultas no tienen conexión directa con el mundo exterior.
- **Capa de salida:** Es la responsable de los cálculos y la transferencia de información de la red a la función externa (entorno).

Las redes de perceptrón multicapa (MLP) generalmente se usan para el formato de aprendizaje supervisado. Un algoritmo de entrenamiento típico para redes MLP es el algoritmo de retropropagación.

Sin embargo, el mejor ejemplo para ilustrar el perceptrón de capa única es a través de la representación de "Regresión logística".



Los **pasos básicos** para entrenar la regresión logística serán los siguientes:

- Los pesos se inicializan con valores aleatorios al comienzo del entrenamiento.
- Para cada elemento del conjunto de entrenamiento, el error se calcula con la diferencia entre la salida deseada y la salida real. El error calculado se utiliza para ajustar los pesos.
- El proceso se repite hasta que el error cometido en todo el conjunto de entrenamiento no sea inferior al umbral especificado, o hasta que se alcance el número máximo de iteraciones.

La regresión logística se considera como un análisis predictivo, se usa para describir datos y explicar la relación entre una variable binaria dependiente y una o más variables nominales o independientes.

Ejemplo de perceptrón

Vamos a desarrollar un algoritmo que clasifique los valores en dos categorías: p1 y p2.

Lo primero que haremos será crear una clase llamada “Perceptron”, para definir los atributos del Perceptrón:

- Datos de entrenamiento.
- Salida esperada.
- Otras características.
- class Perceptron:
- def __init__(self, sample, exit, learn_rate=0.01, epoch_number=1000, bias=-1):
- self.sample = sample
- self.exit = exit
- self.learn_rate = learn_rate
- self.epoch_number = epoch_number
- self.bias = bias
- self.number_sample = len(sample)
- self.col_sample = len(sample[0])
- self.weight = []

Declaramos un método dentro de la clase “Perceptron”, llamado “training” para entrenar a la

```
neurona.

def trainig(self):

for sample in self.sample:

sample.insert(0, self.bias)

for i in range(self.col_sample):

self.weight.append(random.random())

self.weight.insert(0, self.bias)

epoch_count = 0

while True:

erro = False

for i in range(self.number_sample):

u = 0

for j in range(self.col_sample + 1):

u = u + self.weight[j] * self.sample[i][j]

y = self.sign(u)

if y != self.exit[i]:

for j in range(self.col_sample + 1):

self.weight[j] = self.weight[j] + self.learn_rate * (self.exit[i] - y) * self.sample[i][j]

erro = True

epoch_count = epoch_count + 1

if erro == False:
```

```
print(('\nEpoch:\n',epoch_count))

print('------------------------\n')

break
```

El **método "sort"** recibe como argumentos los **datos** que la neurona utilizará para su entrenamiento, este método clasificará los nuevos datos con respecto a sus conocimientos.

```
def sort(self, sample):

sample.insert(0, self.bias)

u = 0

for i in range(self.col_sample + 1):

u = u + self.weight[i] * sample[i]

y = self.sign(u)

if y == -1:

print(('Ejemplo: ', sample))

print('Clasificación: P1')

else:

print(('Ejemplo: ', sample))

print('Clasificación: P2')
```

También tenemos una función de apoyo para calcular el signo de un dato.

```
def sign(self, u):

return 1 if u >= 0 else -1
```

Cargamos los datos de ejemplo y las salidas. Se trata de los datos que la neurona aprenderá y el

conocimiento que utilizará para clasificar datos desconocidos.

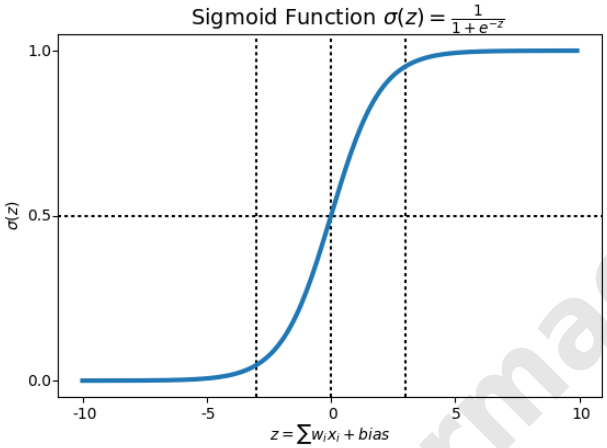
```
samples = [

[1, 4],

[5, 7],

[1, 3],

[6, 9],

[1,2],

[2,1],

[8,4],

[9,4],

[6,8],

]

exit = [-1, 1, -1, 1, -1, -1, 1, 1, 1]
```

Finalmente, utilizamos la clase **"Perceptron"**, creando una instancia **network**, y llamamos al método "**training**". El programa solicita por teclado dos datos nuevos al usuario para que la neurona clasifique.

```
network = Perceptron(sample=samples, exit = exit, learn_rate=0.01, epoch_number=1000, bias=-1)

network.trainig()

while True:

sample = []

for i in range(2):
```

```
sample.insert(i, float(input('Valor: ')))

network.sort(sample)
```

Probando el **programa en Sypder**, obtenemos la siguiente salida:



Vemos que el resultado depende del conocimiento que la neurona ha adquirido anteriormente. Se prueba para los valores 2, 3 y para 7, 8.

A continuación, vemos todo el **código del programa**.

```
import random

class Perceptron:

def __init__(self, sample, exit, learn_rate=0.01, epoch_number=1000, bias=-1):

self.sample = sample

self.exit = exit

self.learn_rate = learn_rate

self.epoch_number = epoch_number

self.bias = bias

self.number_sample = len(sample)
```

```
self.col_sample = len(sample[0])

self.weight = []

def trainig(self):

for sample in self.sample:

sample.insert(0, self.bias)

for i in range(self.col_sample):

self.weight.append(random.random())

self.weight.insert(0, self.bias)

epoch_count = 0

while True:

erro = False

for i in range(self.number_sample):

u = 0

for j in range(self.col_sample + 1):

u = u + self.weight[j] * self.sample[i][j]

y = self.sign(u)

if y != self.exit[i]:

for j in range(self.col_sample + 1):

self.weight[j] = self.weight[j] + self.learn_rate * (self.exit[i] - y) * self.sample[i][j]

erro = True

epoch_count = epoch_count + 1
```

```
if erro == False:

print(('\nEpoch:\n',epoch_count))

print('------------------------\n')

break

def sort(self, sample):

sample.insert(0, self.bias)

u = 0

for i in range(self.col_sample + 1):

u = u + self.weight[i] * sample[i]

y = self.sign(u)

if y == -1:

print(('Ejemplo: ', sample))

print('Clasificación: P1')

else:

print(('Ejemplo: ', sample))

print('Clasificación: P2')

def sign(self, u):

return 1 if u >= 0 else -1

samples = [

[1, 4],

[5, 7],
```

```
[1, 3],

[6, 9],

[1,2],

[2,1],

[8,4],

[9,4],

[6,8],

]

exit = [-1, 1, -1, 1, -1, -1, 1, 1, 1]

network = Perceptron(sample=samples, exit = exit, learn_rate=0.01, epoch_number=1000, bias=-1)

network.trainig()

while True:

sample = []

for i in range(2):

sample.insert(i, float(input('Valor: ')))

network.sort(sample)
```

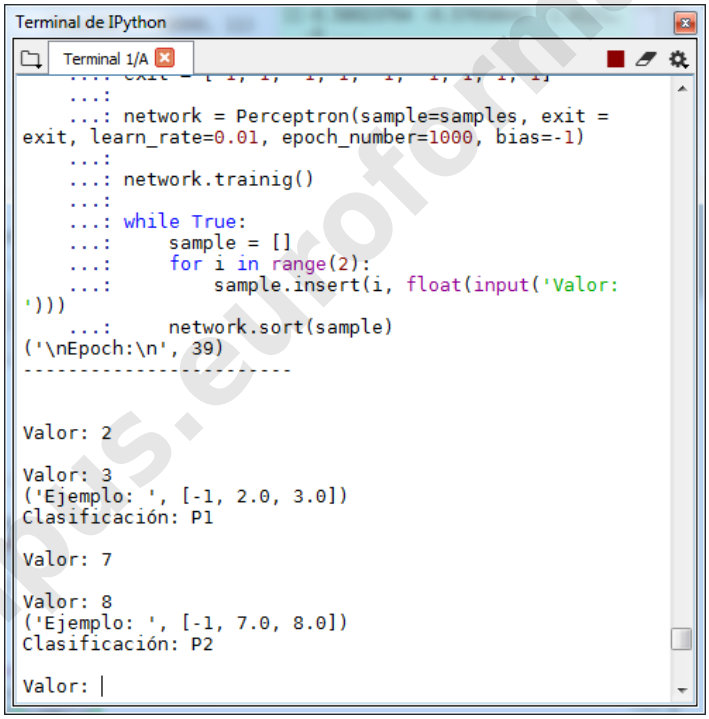
4.3.3. Redes multicapa

Tenemos que decidir si estamos construyendo un **clasificador** o si estamos tratando de encontrar patrones en los datos y si vamos a utilizar el aprendizaje no supervisado. Por ejemplo, para **extraer patrones** de un conjunto de datos no etiquetados, utilizamos una máquina de Boltzman restringida o un codificador automático.

Consideremos los siguientes puntos al elegir una red profunda:

- Para el procesamiento de texto, análisis de sentimientos, análisis y reconocimiento de entidades de nombre, utilizamos una red de tensor neural recurrente neta recursiva o RNTN.
- Para cualquier modelo de lenguaje que opera a nivel de caracteres, usamos la red recurrente.
- Para el reconocimiento de imágenes, utilizamos la red de creencias profundas DBN (Deep Belief Network) o una red convolucional.
- Para el reconocimiento de objetos, utilizamos un RNTN o una red convolucional.
- Para el reconocimiento de voz, utilizamos la red recurrente.

En general, las redes de **creencias profundas** y los **perceptrones** multicapa con unidades lineales rectificadas (Rectified Linear Units - RELU) son buenas opciones para la clasificación. Para el análisis de series de tiempo, siempre se recomienda utilizar la red recurrente.



Tipos de redes neuronales

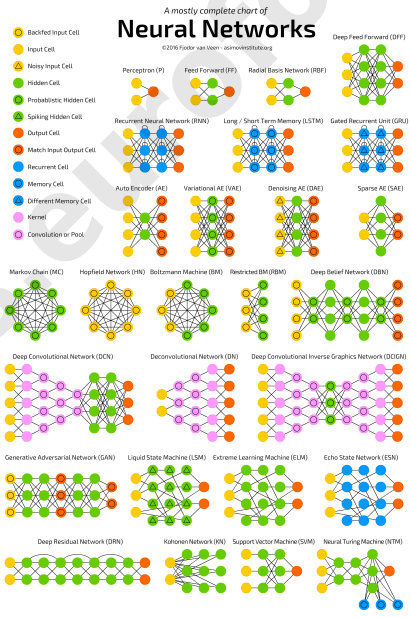
Las redes neuronales han existido por más de 50 años, pero solamente ahora es cuando se están utilizando con profusión. La razón es que son difíciles de entrenar; Cuando tratamos de entrenarlas con un método llamado retro propagación, nos encontramos con un problema llamado gradientes que desaparecen o explotan. Cuando eso sucede, el entrenamiento lleva más tiempo y la precisión queda en segundo plano.

Al entrenar un conjunto de datos, estamos constantemente calculando la **función de coste**, que es la diferencia entre la producción predicha y la producción real de un conjunto de datos de entrenamiento etiquetados. La función de coste se minimiza ajustando los valores de pesos y sesgos hasta que el valor más bajo es obtenido. El proceso de entrenamiento utiliza un gradiente, que es la velocidad a la que cambiará el coste con respecto al cambio en los valores de peso o sesgo.

Redes de Boltzman restringidas o codificadores automáticos - RBN

En 2006, se logró un gran avance al abordar el problema de la desaparición de los gradientes. Geoff Hinton ideó una estrategia novedosa que condujo al desarrollo de la máquina restringida de Boltzman: **RBM**, una red poco profunda de dos capas.

La primera capa es la **capa visible** y la segunda capa es la **capa oculta**. Cada nodo en la capa visible está conectado a cada nodo en la capa oculta. La red se conoce como restringida ya que no se permite que dos capas dentro de la misma capa compartan una conexión.



Los **autoencoders** son redes que codifican datos de entrada como vectores. Crean una representación oculta o comprimida de los datos sin procesar. Los vectores son útiles en la reducción de dimensionalidad; El vector comprime los datos sin procesar en un número menor de dimensiones esenciales. Los codificadores automáticos se combinan con decodificadores, lo que permite la reconstrucción de datos de entrada en función de su representación oculta.

RBM es el equivalente matemático de un traductor bidireccional. Un paso hacia adelante toma entradas y las traduce en un conjunto de números que codifica las entradas. Mientras tanto, un paso hacia atrás toma este conjunto de números y los traduce nuevamente en entradas reconstruidas. Una red bien entrenada realiza back propagation con un alto grado de precisión.

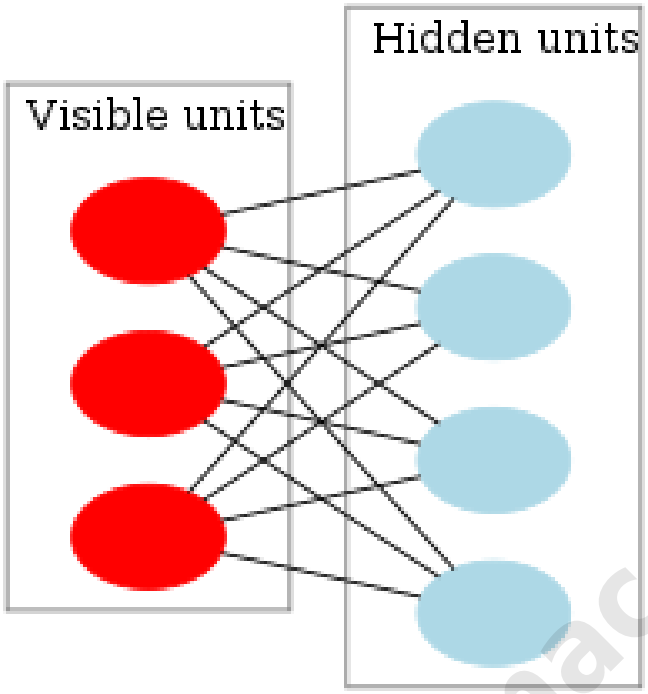
En cualquiera de los pasos, los pesos y los sesgos tienen un papel crítico; ayudan al **RBM** a decodificar las interrelaciones entre las entradas y a decidir qué entradas son esenciales para detectar patrones. A través de pasos hacia adelante y hacia atrás, el RBM está entrenado para reconstruir la entrada con diferentes pesos y sesgos hasta que la entrada y la construcción estén lo más cerca posible.

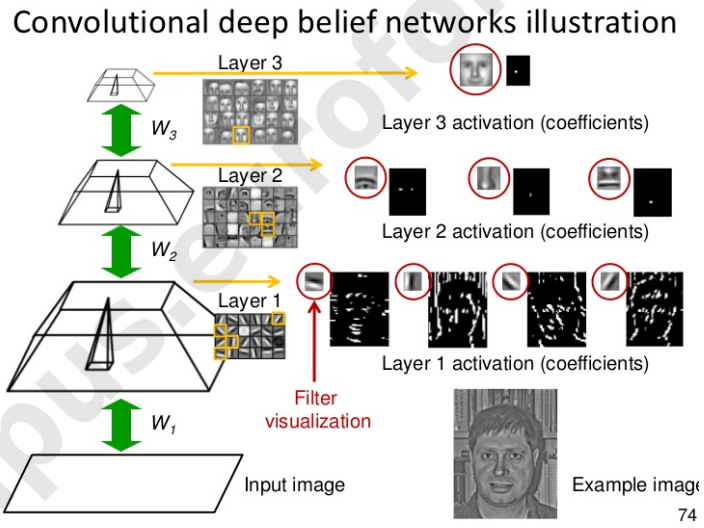
Un aspecto interesante de RBM es que **los datos no necesitan ser etiquetados**. Esto resulta ser muy importante para los conjuntos de datos del mundo real como fotos, videos, voces y datos de sensores, todos los cuales tienden a estar sin etiquetar.

En lugar de etiquetar manualmente los datos por humanos, RBM clasifica automáticamente los datos; ajustando adecuadamente los pesos y sesgos, un RBM puede extraer características importantes y reconstruir la entrada. RBM es parte de la familia de redes neuronales de extracción de características, que están diseñados para reconocer patrones inherentes en los datos. También se denominan codificadores automáticos porque tienen que codificar su propia estructura.

Redes de creencias profundas - DBN

Las redes de creencias profundas (DBN) se forman combinando RBM e introduciendo un método de entrenamiento inteligente. Tenemos un nuevo modelo que finalmente resuelve el problema de la desaparición del gradiente. Geoff Hinton inventó los RBM y también las Redes de creencias profundas como alternativa a la propagación inversa.





Un DBN es similar en estructura a un MLP (perceptrón multicapa), pero muy diferente cuando se trata de entrenamiento. Es la capacitación que permite a los DBN superar a sus homólogos poco profundos.

Un DBN se puede visualizar como una pila de RBM donde la capa oculta de un RBM es la capa visible del RBM por encima de él. El primer RBM está entrenado para reconstruir su entrada con la mayor precisión posible.

La capa oculta del primer RBM se toma como la capa visible del segundo RBM y el segundo RBM se

entrena utilizando las salidas del primer RBM. Este proceso se repite hasta que se entrena cada capa de la red.

En un **DBN**, cada **RBM** aprende la entrada completa. Un DBN funciona globalmente ajustando la entrada completa en sucesión a medida que el modelo mejora lentamente como una lente de cámara enfocando lentamente una imagen. En cuanto al rendimiento, una pila de RBM supera a un solo RBM como un perceptrón multicapa MLP supera a un solo perceptrón.

En esta etapa, los RBM han detectado patrones inherentes en los datos, pero sin ningún nombre o etiqueta. Para finalizar el entrenamiento de la DBN, tenemos que introducir etiquetas en los patrones y ajustar la red con aprendizaje supervisado.

Necesitamos un conjunto muy pequeño de muestras etiquetadas para que las características y los patrones puedan asociarse con un nombre. Este conjunto de datos con etiqueta pequeña se utiliza para el entrenamiento. Este conjunto de datos etiquetados puede ser muy pequeño en comparación con el conjunto de datos original.

Los pesos y sesgos se alteran ligeramente, lo que resulta en un pequeño cambio en la percepción de la red de los patrones y, a menudo, un pequeño aumento en la precisión total.

El entrenamiento también se puede completar en un período de tiempo razonable mediante el uso de GPU que brindan resultados muy precisos en comparación con las redes poco profundas y también con estas redes tenemos una solución para el problema del gradiente de fuga.

Redes Adversarias Generativas - GANs

Las redes adversarias generativas son redes neuronales profundas que comprenden dos redes, enfrentadas una contra la otra, de ahí el nombre de "adversarias".

Las GAN se introdujeron en un artículo publicado por investigadores de la Universidad de Montreal en 2014. El experto en inteligencia artificial de Facebook, Yann LeCun, refiriéndose a las GAN, calificó el entrenamiento de confrontación como "la idea más interesante en los últimos 10 años en ML".

El potencial de GAN es enorme, ya que el escaneo de red aprende a imitar cualquier distribución de datos. Se puede enseñar a las GAN a crear mundos paralelos sorprendentemente similares al

nuestro en cualquier dominio: imágenes, música, discurso, prosa. En cierto modo, son artistas robots, y su producción es bastante impresionante.

En una GAN, una red neuronal, conocida como el **generador**, genera nuevas instancias de datos, mientras que la otra, el **discriminador**, las evalúa para verificar su autenticidad.

Digamos que estamos tratando de generar números escritos a mano como los que se encuentran en el conjunto de datos MNIST, que se toma del mundo real. El trabajo del discriminador, cuando se muestra una instancia del verdadero conjunto de datos MNIST, es reconocerlos como auténticos.

Consideremos los siguientes pasos de la **GAN**:

La red del generador toma la entrada en forma de números aleatorios y devuelve una imagen.

Esta imagen generada se proporciona como entrada a la red discriminadora junto con un flujo de imágenes tomadas del conjunto de datos real.

El discriminador toma imágenes reales y falsas y devuelve probabilidades, un número entre 0 y 1, donde 1 representa una predicción de autenticidad y 0 representa falso.

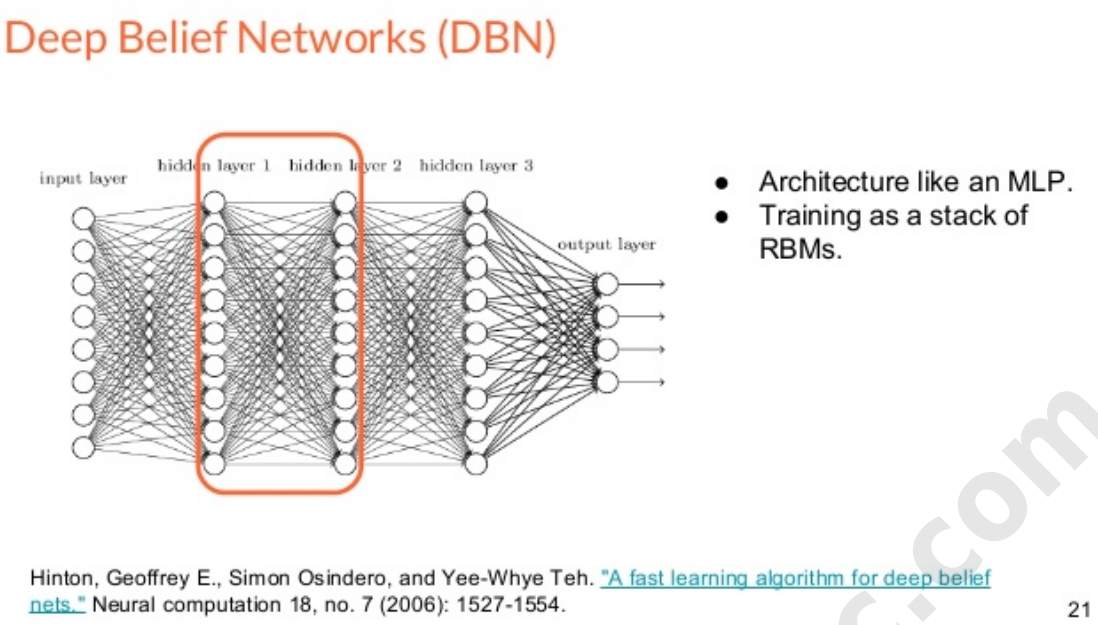
Entonces tenemos un **circuito de retroalimentación doble**:

El discriminador está en un circuito de retroalimentación con la verdad básica de las imágenes, que conocemos.

El generador está en un circuito de retroalimentación con el discriminador.

Redes neuronales recurrentes - RNN

Son redes neuronales en las que los datos pueden fluir en cualquier dirección. Estas redes se utilizan para aplicaciones como el modelado de idiomas o el procesamiento del lenguaje natural (PNL).



El concepto básico subyacente a los RNN es utilizar información secuencial. En una red neuronal normal se supone que todas las entradas y salidas son independientes entre sí. Si queremos predecir la siguiente palabra en una oración, tenemos que saber qué palabras vinieron antes.

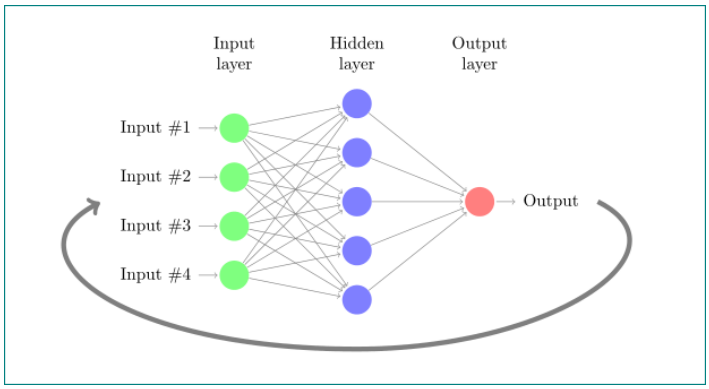
Los RNN se denominan recurrentes ya que repiten la misma tarea para cada elemento de una secuencia, y la salida se basa en los cálculos anteriores. Por lo tanto, se puede decir que los RNN tienen una "memoria" que captura información sobre lo que se ha calculado previamente. En teoría, los RNN pueden usar la información en secuencias muy largas, pero en realidad, solo pueden mirar hacia atrás unos pocos pasos.

Las redes de memoria a largo plazo (LSTM) son las RNN más utilizadas.

Junto con las redes neuronales convolucionales, los RNN se han utilizado como parte de un modelo para generar descripciones de imágenes no etiquetadas, con un gran rendimiento.

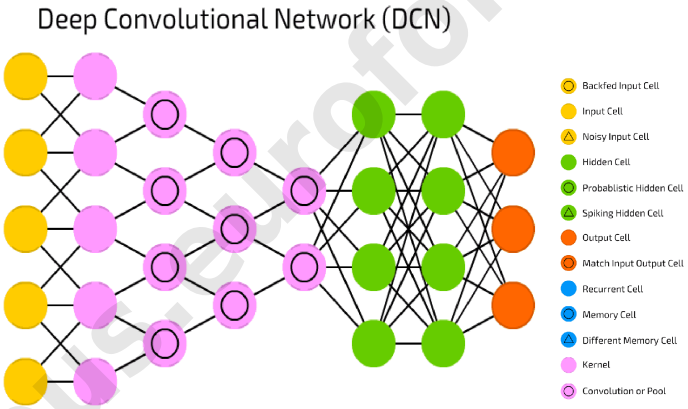
Redes neuronales profundas convolucionales - CNN

Si aumentamos el número de capas en una red neuronal para profundizarla, aumenta la complejidad de la red y nos permite modelar funciones que son más complicadas. Sin embargo, el número de pesos y sesgos aumentará exponencialmente. De hecho, aprender problemas tan difíciles puede volverse imposible para las redes neuronales normales. Esto lleva a una solución, las redes neuronales convolucionales.



Las CNN se usan ampliamente en visión artificial; se han aplicado también en modelado acústico para reconocimiento automático de voz.

La idea detrás de las redes neuronales convolucionales es la idea de un "filtro en movimiento" que pasa a través de la imagen. Este filtro móvil, o convolución, se aplica a una determinada vecindad de nodos que, por ejemplo, pueden ser píxeles, donde el filtro aplicado es 0.5 x el valor del nodo.



El destacado **investigador Yann LeCun** fue pionero en las redes neuronales convolucionales. Facebook, como software de reconocimiento facial, utiliza estas redes. CNN ha sido la solución para proyectos de visión artificial. Hay muchas capas en una red convolucional. En el desafío de Imagenet, una máquina pudo vencer a un humano en el reconocimiento de objetos en 2015.

En pocas palabras, las redes neuronales convolucionales (CNN) son redes neuronales de múltiples capas. Las capas son a veces hasta 17 o más y asumen que los datos de entrada son imágenes.

Las CNN reducen drásticamente la cantidad de parámetros que deben ajustarse. Por lo que manejan eficientemente la alta dimensionalidad de las imágenes en bruto.

Trabajar con TensorFlow y Python

TensorFlow es una biblioteca o marco de software, diseñado por el equipo de Google para implementar conceptos de aprendizaje automático y aprendizaje profundo de la manera más fácil. Combina el álgebra computacional de las técnicas de optimización para facilitar el cálculo de muchas expresiones matemáticas.

Consideremos ahora las siguientes características importantes de **TensorFlow**:

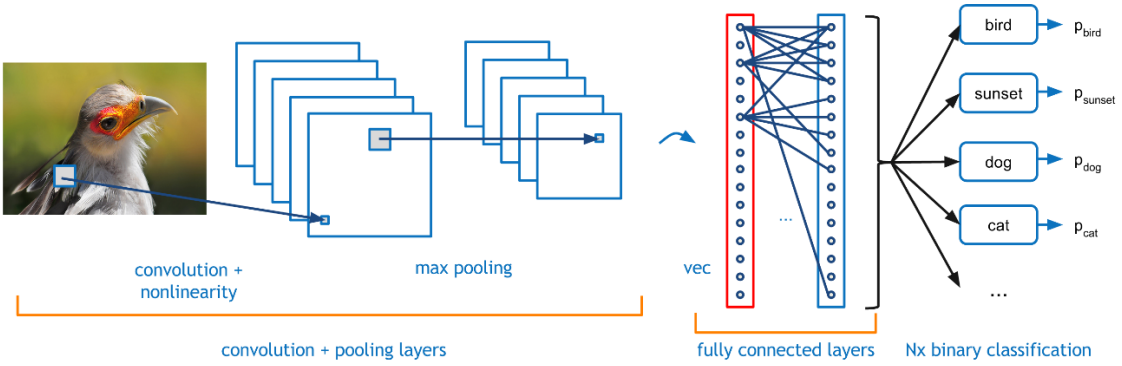
- Define, optimiza y calcula expresiones matemáticas fácilmente con la ayuda de matrices multidimensionales llamadas tensores.
- Incluye un soporte de programación de redes neuronales profundas y técnicas de aprendizaje automático.
- Incluye una característica de computación altamente escalable con varios conjuntos de datos.
- TensorFlow utiliza la informática GPU, automatizando la gestión. También incluye una característica única de optimización de la misma memoria y los datos utilizados.

La razón de la popularidad de TensorFlow es que está bien documentado e incluye muchas bibliotecas de aprendizaje automático. Ofrece algunas funcionalidades y métodos importantes para ello.

TensorFlow también se llama un **producto** "**Google**". Incluye una variedad de algoritmos de aprendizaje automático y aprendizaje profundo. Puede entrenar y ejecutar redes neuronales profundas para la clasificación de dígitos escritos a mano, reconocimiento de imágenes, incrustación de palabras y creación de varios modelos de secuencia.

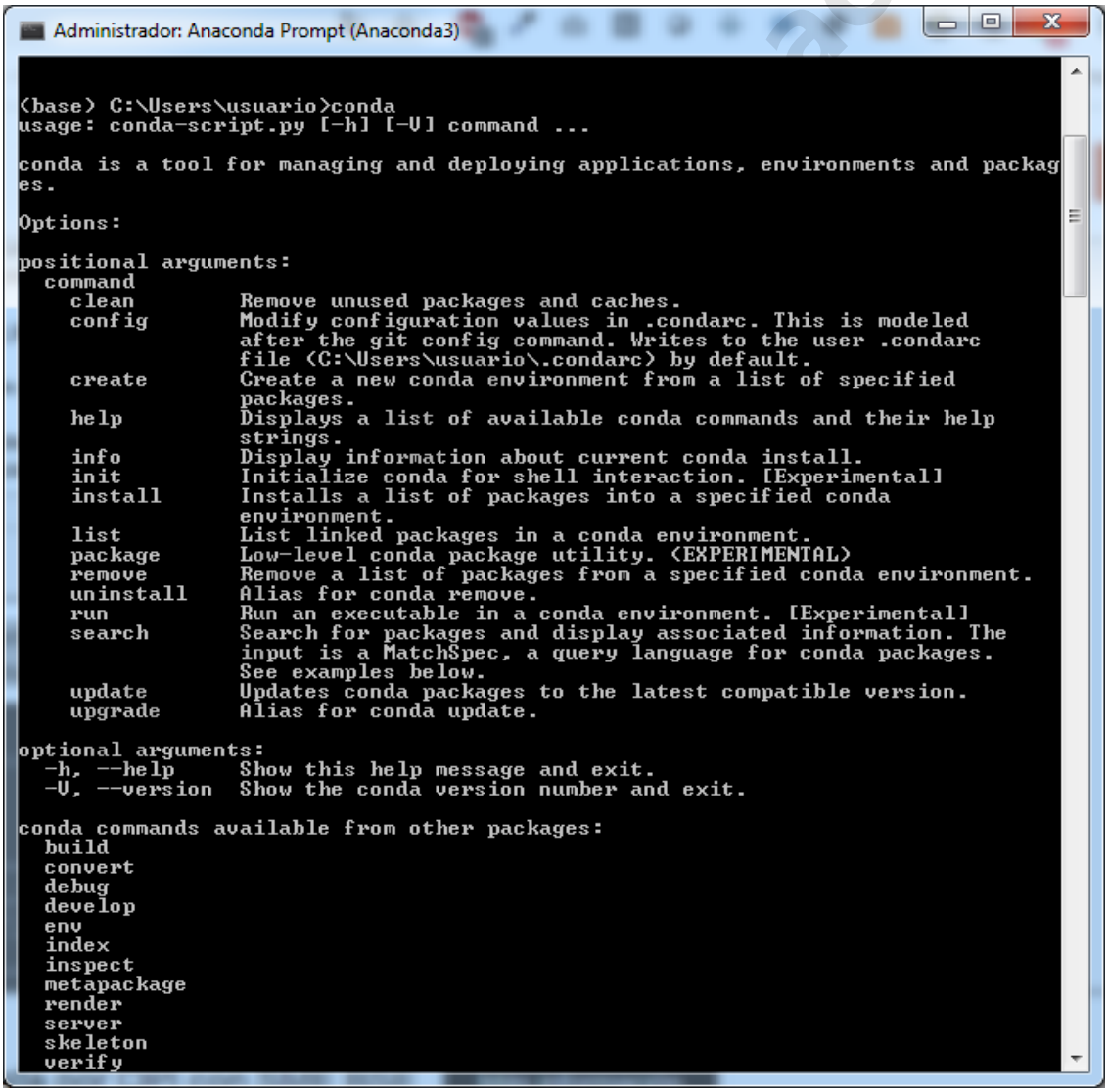
Como ya hemos visto, la instalación de TensorFlow se puede hacer a través de Conda. Con Conda, podemos crear, exportar, enumerar, eliminar y actualizar entornos que tengan diferentes versiones de Python y / o paquetes instalados en ellos. Cambiar o moverse entre entornos se llama activar el entorno.

Después de una instalación exitosa, podemos verificar el símbolo del sistema a través del comando "conda". La **ejecución del comando** se muestra a continuación:



Ahora, ejecute el siguiente comando para inicializar la instalación de **TensorFlow**:

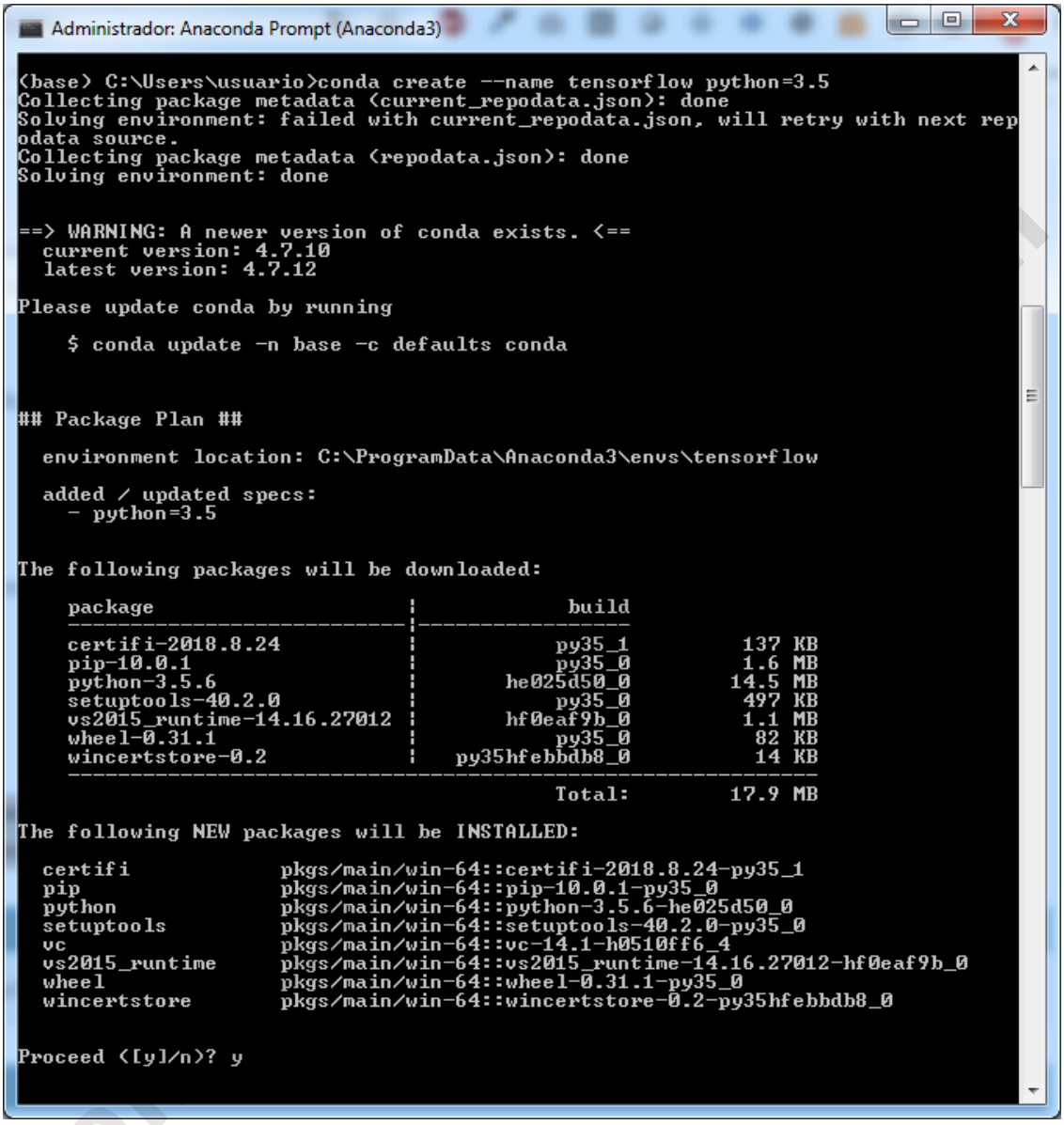
$conda create --name test python=3.6.5



Se descargan los paquetes necesarios para la configuración de TensorFlow.

Después de una configuración ambiental con éxito, es importante activar el módulo TensorFlow.

$activate tensorflow



Vemos como se activa el entorno (tensorflow), con todo lo necesario disponible para nuestros programas.

Ya solo nos quedaría proceder a la **instalación de TensorFlow**, como ya vimos, utilizando pip.

$pip install tensorflow

$pip install tensorflow-gpu

Con el segundo comando se instalan todos los requerimientos gráficos.

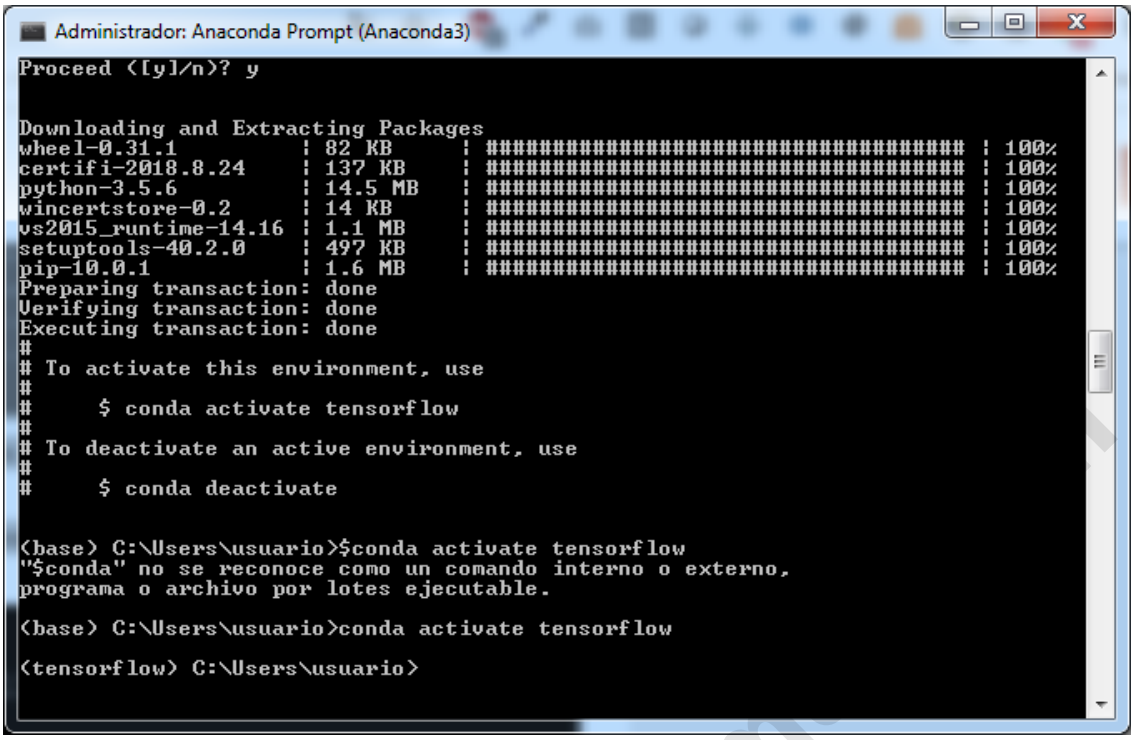
El aprendizaje automático incluye una sección de aprendizaje automático y el aprendizaje profundo es parte del aprendizaje automático. La capacidad del programa que sigue los conceptos de aprendizaje automático es mejorar su rendimiento de los datos observados.

El motivo principal de la transformación de datos es mejorar su conocimiento para lograr mejores resultados en el futuro, proporcionar un resultado más cercano al deseado para ese sistema en particular. El aprendizaje automático incluye el "reconocimiento de patrones", que incluye la capacidad de reconocer los patrones en los datos.

El aprendizaje automático se puede entrenar de dos maneras diferentes:

- Entrenamiento supervisado.
- Entrenamiento sin supervisión.

El **aprendizaje supervisado** o entrenamiento supervisado incluye un procedimiento en el que el conjunto de entrenamiento se proporciona como entrada al sistema en el que cada ejemplo se etiqueta con un valor de salida deseado. El entrenamiento en este tipo se realiza utilizando la minimización de una función de pérdida particular, que representa el error de salida con respecto al sistema de salida deseado.



Después de completar el entrenamiento, la precisión de cada modelo se mide con respecto a ejemplos disjuntos del conjunto de entrenamiento, también llamado conjunto de validación.

El mejor ejemplo para ilustrar el "aprendizaje supervisado" es con un montón de fotos dadas con información incluida en ellas. Aquí, el usuario puede entrenar a un modelo para reconocer nuevas fotos.

En el **aprendizaje no supervisado** o entrenamiento no supervisado, se incluyen ejemplos de entrenamiento que no estén etiquetados por el sistema al que pertenecen. El sistema busca los datos, que comparten características comunes, y los cambia en función de las características de conocimiento interno. Este tipo de algoritmos de aprendizaje se utilizan básicamente en problemas de agrupamiento.

El mejor ejemplo para ilustrar el "aprendizaje no supervisado" es con un montón de fotos sin información incluida y el modelo de entrenamiento de usuarios con clasificación y agrupación. Este tipo de algoritmo de entrenamiento funciona con suposiciones ya que no se proporciona información.

Las **entradas y salidas** se representan como vectores o tensores. Por ejemplo, una red neuronal puede tener las entradas donde los valores individuales de píxeles RGB en una imagen se representan como vectores.

Las capas de neuronas que se encuentran entre la capa de entrada y la capa de salida se denominan **capas ocultas**. Aquí es donde ocurre la mayor parte del trabajo cuando la red neuronal intenta resolver problemas. Un vistazo más de cerca a las capas ocultas puede revelar mucho sobre las características que la red ha aprendido a extraer de los datos.

Se forman diferentes arquitecturas de redes neuronales eligiendo qué neuronas se conectan a las otras neuronas en la siguiente capa.

A continuación, se muestra el pseudocódigo para calcular la salida de la red neuronal de propagación hacia adelante:

node []: = array de nodos ordenados topológicamente

Un borde (edge) de a b significa que a está a la izquierda de b

if la red neuronal tiene entradas R y salidas S,

then los primeros nodos R son nodos de entrada y los últimos **nodos S** son nodos de **salida**.

incoming[x]: = nodos conectados al nodo x

weight[x]: = pesos de los bordes entrantes a x

Para cada neurona x, de izquierda a derecha:

if x <= R: no hacer nada # es un nodo de entrada

inputs [x] = [output[i] para i en incoming [x]]

weighted_sum = dot_product (weights[x], inputs[x])

output[x] = Activation_function (suma ponderada)

4.3.5. Entrenar una red neuronal

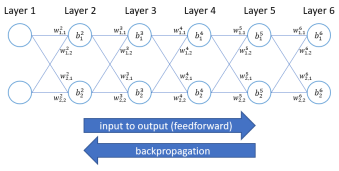
Para entrenar una red neuronal, tenemos que encontrar los valores óptimos de los pesos de una red neuronal para obtener la salida deseada. Para ello utilizamos el método iterativo de descenso de gradiente.

Comenzamos con la inicialización aleatoria de los pesos. Después de la inicialización aleatoria, hacemos predicciones sobre algún subconjunto de datos con el proceso de propagación hacia adelante (feedforward), calculamos la función de coste correspondiente C y actualizamos cada peso w en una cantidad proporcional como dC / dw, es decir, la derivada de las funciones de costo w, r, t (el peso). La constante de proporcionalidad se conoce como **tasa de aprendizaje**.

Los **gradientes** se pueden calcular de manera eficiente utilizando el a**lgoritmo de retropropagación**. La observación clave es que, debido a la regla de diferenciación de la cadena, el gradiente en cada neurona en la red neuronal se puede calcular utilizando el gradiente en las neuronas, tiene bordes salientes. Por lo tanto, calculamos los gradientes hacia atrás, es decir, primero calculamos los gradientes de la capa de salida, luego la capa superior oculta, seguida de la capa oculta anterior, y así sucesivamente, terminando en la capa de entrada.

El algoritmo de retropropagación se implementa principalmente utilizando la idea de un gráfico computacional (que veremos a continuación), donde cada neurona se expande a muchos nodos en el gráfico computacional y realiza una operación matemática simple como la suma o la multiplicación. El gráfico computacional no tiene ningún peso en los bordes.

Todos los pesos se asignan a los nodos, por lo que los pesos se convierten en sus propios nodos. El algoritmo de propagación hacia atrás se ejecuta en el gráfico computacional. Una vez que se completa el cálculo, solo se requieren los gradientes de los nodos de peso para la actualización. El resto de los gradientes se pueden descartar.



Técnica de optimización de descenso de gradiente

La función de optimización de uso común que ajusta los pesos de acuerdo con el error que causaron se denomina "descenso de gradiente".

Gradiente es otro nombre para pendiente, y la pendiente, en un gráfico xy, representa cómo se relacionan dos variables entre sí: el aumento sobre el recorrido, el cambio de distancia sobre el

cambio de tiempo, etc. En este caso, la pendiente es la relación entre el error de la red y un solo peso; es decir, cómo cambia el error a medida que varía el peso.

Para decirlo con más precisión, queremos encontrar qué peso produce el menor error. Queremos encontrar el peso que representa correctamente las señales contenidas en los datos de entrada y las traduce a una clasificación correcta.

$$Feed\ forward$$
$$h = \sigma(xw_1 + b)$$
$$y' = \sigma(hw_2 + b)$$

$$Loss\ function$$
$$L = \frac{1}{2}\sum(y - y')^2$$

$$Gradient\ calculation$$
$$\frac{\partial L}{\partial w_2} = \frac{\partial y'}{\partial w_2}\frac{\partial L}{\partial y'}$$
$$\frac{\partial L}{\partial w_1} = ?$$

$$Weight\ update$$
$$w_i^{t+1} \leftarrow w_i^t - \alpha\frac{\partial L}{\partial w_i}$$

A medida que una red neuronal aprende, ajusta lentamente muchos pesos para que puedan asignar

la señal al significado correctamente. La relación entre el error de red y cada uno de esos pesos es una derivada, dE / dw, que calcula el grado en que un ligero cambio en un peso causa un ligero cambio en el error.

Cada peso es solo un factor en una red profunda que involucra muchas transformaciones; la señal del peso pasa a través de activaciones y sumas en varias capas, por lo que utilizamos la regla de cálculo de la cadena para volver a las activaciones y salidas de la red. Esto nos lleva al peso en cuestión y su relación con el error general.

Dadas dos variables, error y peso, están mediadas por una tercera variable, la **activación**, a través de la cual se pasa el peso. Podemos calcular cómo un cambio en el peso afecta un cambio en el error calculando primero cómo un cambio en la activación afecta un cambio en el Error, y cómo un cambio en el peso afecta un cambio en la activación.

La idea básica en el aprendizaje profundo no es más que eso: **ajustar los pesos de un modelo en respuesta al error que produce, hasta que ya no pueda reducir el error**.

La red profunda se entrena lentamente si el valor del gradiente es pequeño y rápido si el valor es alto. Cualquier imprecisión en el entrenamiento conduce a resultados imprecisos. El proceso de entrenar las redes desde la salida hasta la entrada se llama propagación hacia atrás o apoyo hacia atrás. Sabemos que la propagación hacia adelante comienza con la entrada y avanza. La retropropagación hace el inverso u opuesto calculando el gradiente de derecha a izquierda.

Cada vez que calculamos un gradiente, utilizamos todos los gradientes anteriores hasta ese punto.

Comencemos en un nodo en la capa de salida. El borde usa el gradiente en ese nodo. A medida que volvemos a las capas ocultas, se vuelve más complejo. El producto de dos números entre 0 y 1 le da un número menor. El valor del gradiente sigue disminuyendo y, como resultado, la **retropropagación** tarda mucho tiempo para entrenar y la precisión sufre.

Desafíos en los algoritmos de aprendizaje profundo

Hay ciertos desafíos comunes tanto para las redes neuronales poco profundas como para las redes neuronales profundas, como el sobreajuste y el tiempo de cálculo. Los DNN (Deep Neural Network) se ven afectados por el sobreajuste debido al uso de capas adicionales de abstracción que les

permiten modelar dependencias raras en los datos de entrenamiento.

Los métodos de **regularización**, como la deserción (dropout), la detención temprana, el aumento de datos, o el aprendizaje de transferencia se aplican durante el entrenamiento para combatir el sobreajuste.

La **regularización de abandono (dropout)** omite aleatoriamente unidades de las capas ocultas durante el entrenamiento, lo que ayuda a evitar dependencias raras. Los DNN tienen en cuenta varios parámetros de entrenamiento, como el tamaño, es decir, el número de capas y el número de unidades por capa, la tasa de aprendizaje y los pesos iniciales.

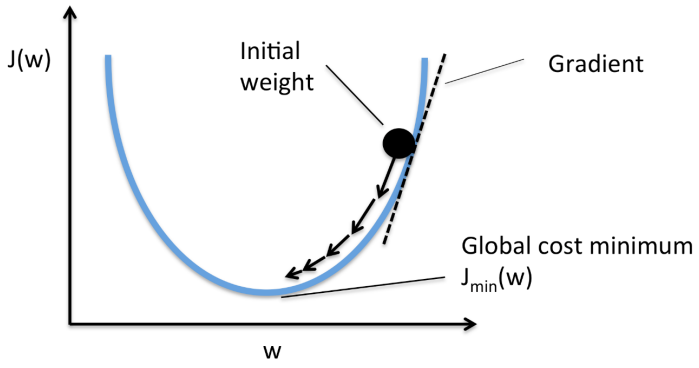
Encontrar parámetros óptimos no siempre es práctico debido al alto coste de tiempo y recursos computacionales. Varios trucos, como el procesamiento por lotes, pueden acelerar el cálculo. El gran poder de procesamiento de las GPU ha ayudado significativamente al proceso de entrenamiento, ya que la matriz y los cálculos vectoriales requeridos están bien ejecutados en las GPU.

Dropout

La deserción es una técnica de regularización popular para redes neuronales. Las redes neuronales profundas son particularmente propensas al sobreajuste.

En palabras de Geoffrey Hinton, uno de los pioneros de Deep Learning, “Si tienes una red neuronal profunda y no está sobreajustada, probablemente deberías estar usando una más grande y dropout”.

La **deserción** es una **técnica** en la que, durante cada **iteración de descenso de gradiente**, dejamos caer un conjunto de nodos seleccionados al azar. Esto significa que ignoramos algunos nodos al azar como si no existieran.



Cada neurona se mantiene con una probabilidad de q y se cae al azar con probabilidad 1-q. El valor q puede ser diferente para cada capa en la red neuronal. Un valor de 0.5 para las capas ocultas y 0 para la capa de entrada funciona bien en una amplia gama de tareas.

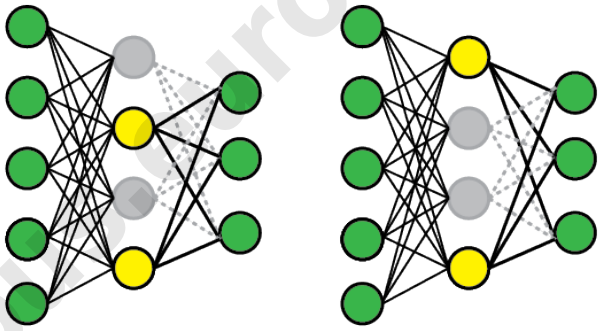
Durante la evaluación y predicción, no se utiliza abandono. La salida de cada neurona se multiplica por q para que la entrada a la siguiente capa tenga el mismo valor esperado.

La idea detrás del dropout es la siguiente: en una red neuronal sin regularización del abandono, las neuronas desarrollan una codependencia entre ellas que conduce al sobreajuste.

El dropout se implementa con bibliotecas como TensorFlow y Pytorch manteniendo la salida de las neuronas seleccionadas al azar como 0. Es decir, aunque la neurona existe, su salida se sobrescribe como 0.

Parar temprano (Early Stopping)

Entrenamos redes neuronales usando un algoritmo iterativo llamado descenso de gradiente.



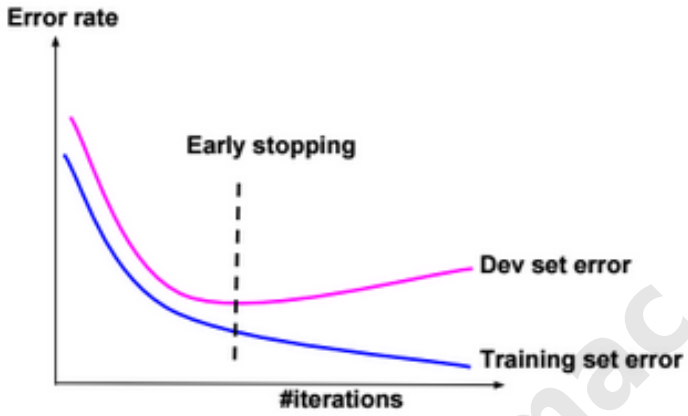
La idea detrás de la detención temprana es intuitiva; Dejamos de entrenar cuando el error comienza a aumentar. Aquí, por error, nos referimos al error medido en los datos de validación, que es la parte de los datos de entrenamiento utilizados para ajustar los hiperparámetros. En este caso, el hiperparámetro es el criterio de detención.

Aumento de datos (Data Augmentation)

Es un proceso en el que aumentamos la cantidad de datos que tenemos o los aumentamos utilizando los datos existentes y aplicando algunas transformaciones. Las transformaciones exactas utilizadas dependen de la tarea que pretendemos lograr. Además, las transformaciones que ayudan a la red

neuronal dependen de su arquitectura.

Por ejemplo, en muchas tareas de visión por computadora, como la clasificación de objetos, una técnica efectiva de aumento de datos agrega nuevos puntos de datos que son versiones recortadas o traducidas de datos originales.



Cuando una computadora acepta una imagen como entrada, toma una matriz de valores de píxeles. Digamos que toda la imagen se desplaza 15 píxeles hacia la izquierda. Aplicamos muchos cambios diferentes en diferentes direcciones, lo que resulta en un conjunto de datos aumentado muchas veces el tamaño del conjunto de datos original.

Transferir aprendizaje

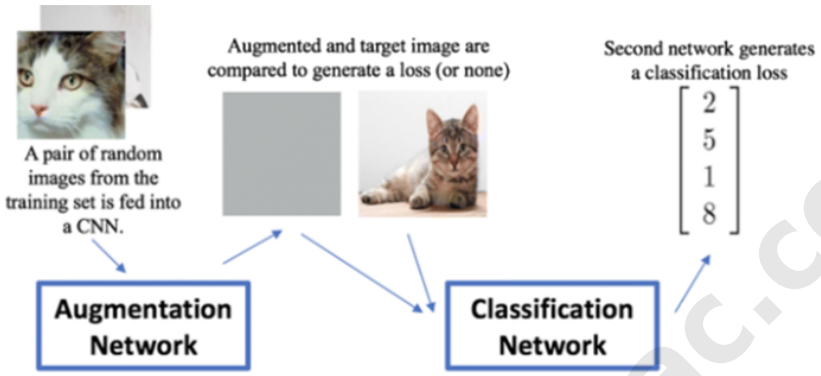
El proceso de tomar un modelo pre-entrenado y "ajustar" el modelo con nuestro propio conjunto de datos se llama aprendizaje de transferencia. Hay varias formas de hacer esto. Algunas se describen a continuación:

- Entrenamos el modelo pre-entrenado en un gran conjunto de datos. Luego, eliminamos la última capa de la red y la reemplazamos con una nueva capa con pesos aleatorios.
- Luego congelamos los pesos de todas las otras capas y entrenamos la red normalmente. Aquí congelar las capas no está cambiando los pesos durante el descenso o la optimización del gradiente.

El concepto detrás de esto es que el modelo pre-entrenado actuará como un extractor de características, y solo la última capa será entrenada en la tarea actual.

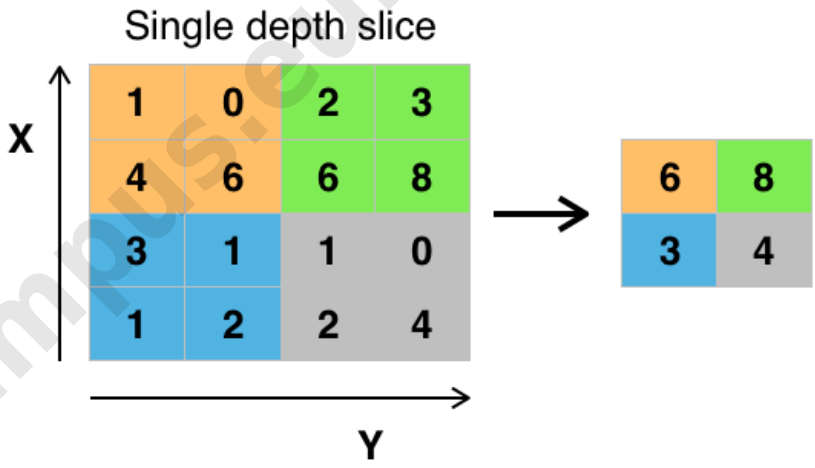
Max Pooling

La agrupación máxima es un proceso de **discretización** basado en muestras. El objetivo es reducir la muestra de una representación de entrada, lo que reduce la dimensionalidad con los supuestos requeridos.



Memoria a corto y largo plazo (LSTM)

LSTM controla la decisión sobre qué entradas deben tomarse dentro de la neurona especificada. Incluye el control para decidir qué se debe calcular y qué salida se debe generar.



Implementación de una red profunda

En esta implementación de Deep learning, nuestro objetivo es predecir la rotación de clientes para un determinado banco, y de esta forma saber que clientes tienen mayor probabilidad de abandonar este servicio bancario.

El conjunto de datos utilizado es relativamente pequeño y contiene 10000 filas con 14 columnas.

El archivo con los datos utilizado es **Churn_Modelling.csv**, y se puede descargar de la sección de recursos del curso.

Descargar Documento.

Estamos utilizando la **distribución Anaconda** y marcos como Theano, TensorFlow y Keras. Keras está construido sobre Tensorflow y Theano, que funcionan como backends.

Comenzamos instalando estas librerías y cargando los datos:

```
# Artificial Neural Network

# Instalar Theano

pip install --upgrade theano

# Instalar Tensorflow

pip install --upgrade tensorflow

# Instalar Keras

pip install --upgrade keras

# Importar las librerias

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importar la base de datos

dataset = pd.read_csv('c:/python/Churn_Modelling.csv')
```

Creamos arrays con las características (valores) incluidas en el conjunto de datos y la variable objetivo, que es la columna 14, etiquetada como "Exited".

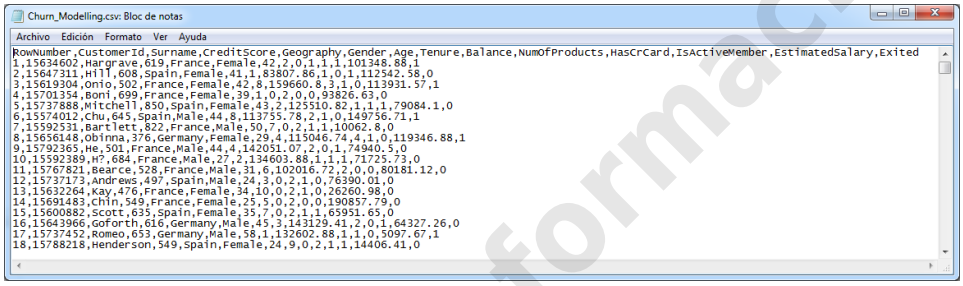
El **aspecto inicial de los datos** es como se muestra a continuación:

X = dataset.iloc[:, 3:13].values

Y = dataset.iloc[:, 13].values

X

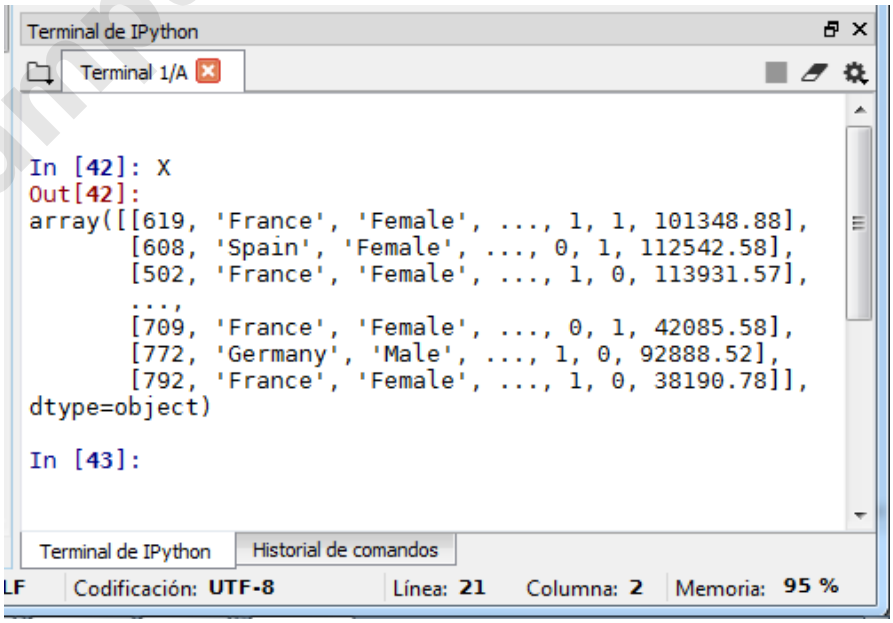
Al mostrar “X” obtenemos el siguiente resultado:



En Y tenemos los valores de la variable “Exited”:

Y

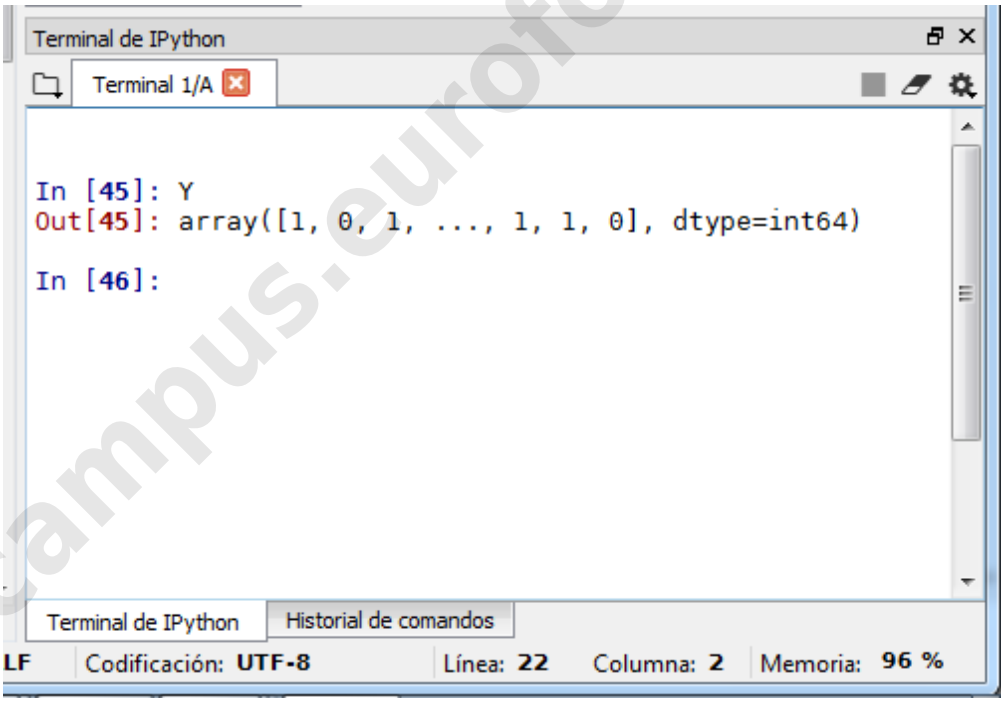
Obtenemos los valores de Y:



Para hacer más real el análisis lo simplificamos codificando las variables de texto. Estamos utilizando la función ScikitLearn 'LabelEncoder' para codificar automáticamente las diferentes etiquetas en las columnas con valores entre 0 y n_classes-1.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

labelencoder_X_1 = LabelEncoder()

X[:,1] = labelencoder_X_1.fit_transform(X[:,1])

labelencoder_X_2 = LabelEncoder()

X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])

X
```

El resultado es el de los datos de antes pero ya codificados:



En el resultado anterior, los nombres de los países se reemplazan por 0 (Francia), 1 (Alemania) y 2 (España); mientras que masculino y femenino se reemplazan por 0 y 1.

Ahora etiquetamos los datos codificados. Usamos la misma **biblioteca ScikitLearn** y otra función llamada **OneHotEncoder** para simplemente pasar el número de columna creando una variable

ficticia.

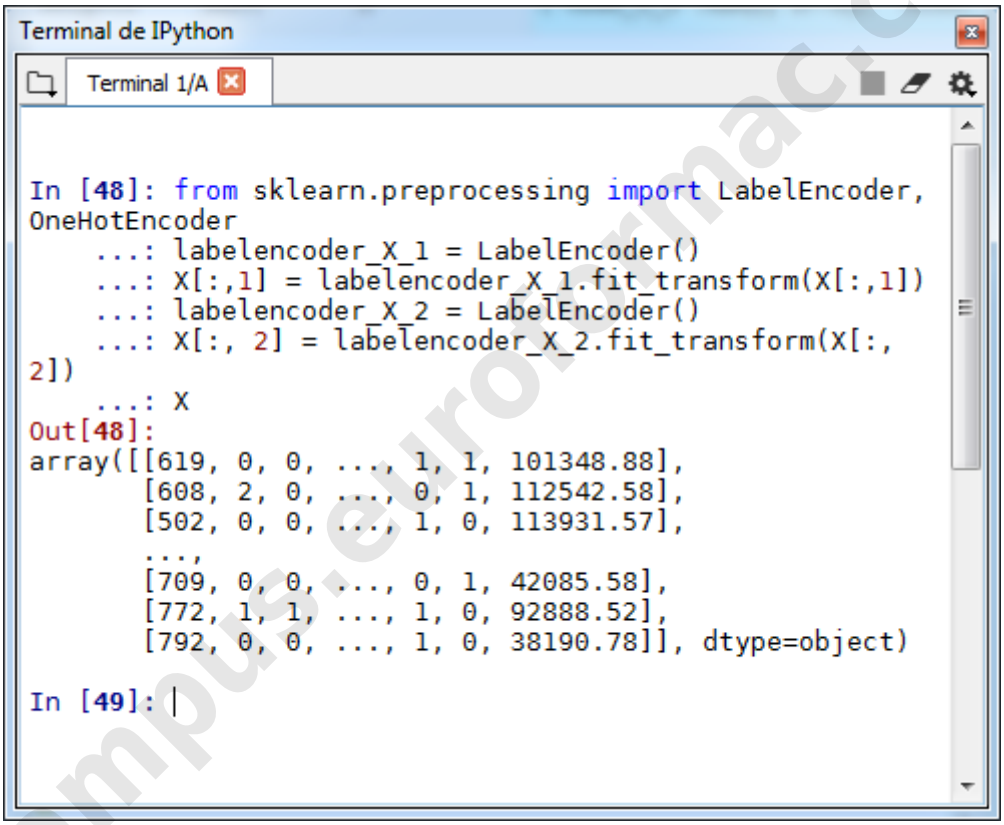
onehotencoder = OneHotEncoder(ColumnTransformer = [1])

X = onehotencoder.fit_transform(X).toarray()

X = X[:, 1:]

X

Ahora, las primeras 2 columnas representan el país y la cuarta columna representa el género.



Siempre dividimos nuestros datos en parte de entrenamiento y pruebas; entrenamos nuestro modelo en datos de entrenamiento y luego verificamos la precisión de un modelo en datos de prueba que ayuda a evaluar la eficiencia del modelo.

Utilizaremos la función train_test_split de ScikitLearn para dividir nuestros datos en conjunto de entrenamiento y conjunto de prueba. Mantenemos el ratio train_test_split a la proporción 80:20.

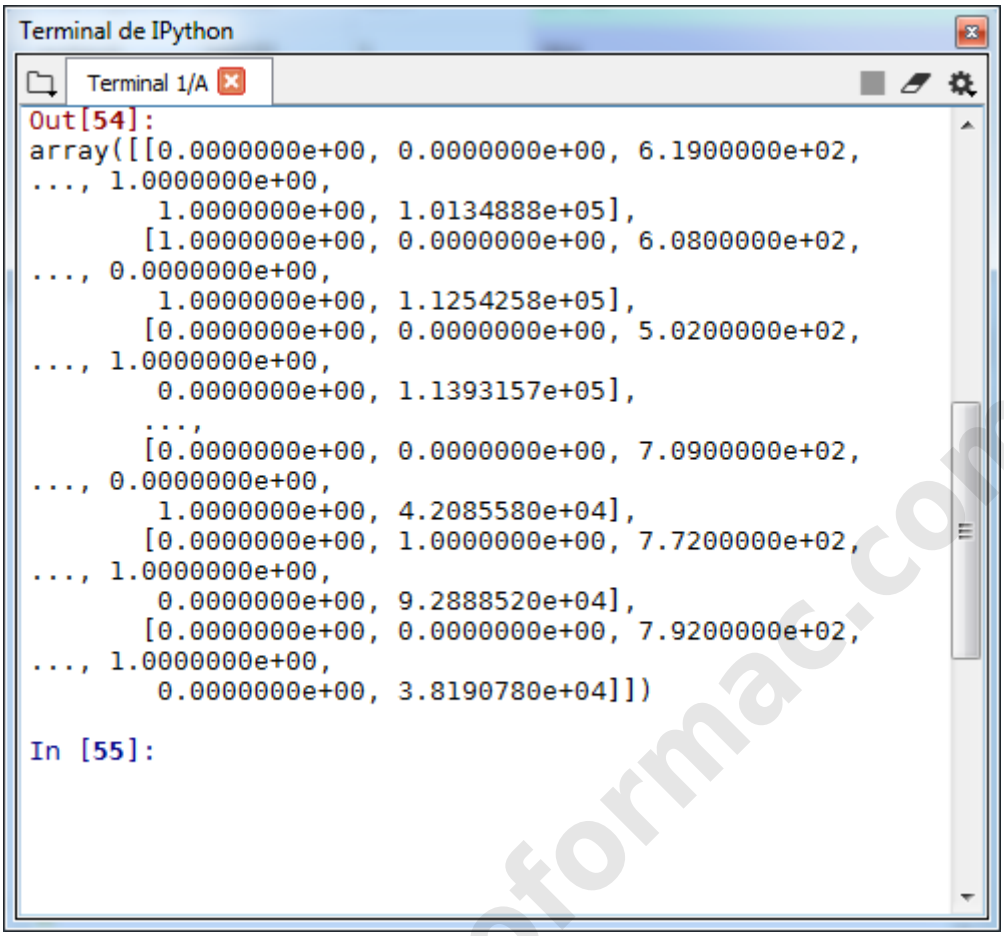
#Dividir el dataset en Training set y Test Set

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.2)
```

Algunas variables tienen valores en miles, mientras que otras tienen valores en decenas o unidades. Escalamos los datos para que sean más representativos.

Ajustamos y transformamos los datos de entrenamiento usando la función StandardScaler . Estandarizamos nuestra escala para que se utilice el mismo método ajustado para transformar / escalar datos de prueba.

```
# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)

X
```



Los datos ahora se escalan correctamente. Finalmente, ya tenemos listo el **preprocesamiento de los datos**. Ahora, comenzaremos con nuestro modelo.

Importamos los módulos necesarios aquí. Necesitamos el módulo secuencial para inicializar la red neuronal y el módulo dense para agregar las capas ocultas.

Importar la libreria de Keras y sus paquetes

import keras

from keras.models import Sequential

from keras.layers import Dense

El nombre del modelo será “Classifier”, ya que nuestro objetivo es clasificar la rotación de clientes. Luego usamos el módulo secuencial para la inicialización.

#Inicializar la red neuronal

classifier = Sequential()

Agregamos las capas ocultas una por una usando la función dense. En el siguiente código, veremos muchos argumentos.

Nuestro primer parámetro es **output_dim**. Es la cantidad de nodos que agregamos a esta capa. **init** es la inicialización del gradiente estocástico descendiente.

En una red neuronal asignamos pesos a cada nodo. En la inicialización, los pesos deben estar cerca de cero y los inicializamos aleatoriamente usando la función **uniform**. El parámetro **input_dim** es necesario solo para la primera capa, ya que el modelo no conoce el número de nuestras variables de entrada. Aquí el número total de variables de entrada es 11. En la segunda capa, el modelo conoce automáticamente el número de variables de entrada de la primera capa oculta.

Las siguientes líneas de **código** agregarán la **capa de entrada** y la primera capa oculta:

classifier.add(Dense(units = 6, kernel_initializer = 'uniform',

activation = 'relu', input_dim = 11))

Agregamos la segunda capa oculta:

classifier.add(Dense(units = 6, kernel_initializer = 'uniform',

activation = 'relu'))

Agregamos la capa de salida:

classifier.add(Dense(units = 1, kernel_initializer = 'uniform',

activation = 'sigmoid'))

Ahora vamos a compilar nuestra red. Hasta ahora hemos agregado varias capas a nuestro clasificador. Ahora los compilaremos utilizando el método de **compile**. Los argumentos agregados en el control de compilación final completan la red neuronal, por lo que debemos ser cuidadosos en este paso.

A continuación, se incluye una breve explicación de los argumentos.

El primer argumento es **Optimizer**. Este es un algoritmo utilizado para encontrar el conjunto óptimo de pesos. Este algoritmo se llama **Descenso de gradiente estocástico (SGD)**. Aquí estamos usando uno entre varios tipos, llamado '**Adam optimizer'**. El SGD depende de la pérdida, por lo que nuestro segundo parámetro es la pérdida.

Si nuestra variable dependiente es binaria, usamos la función de pérdida logarítmica llamada '**binary_crossentropy** ', y si nuestra variable dependiente tiene más de dos categorías en la salida, entonces usamos '**categorical_crossentropy** '. Queremos mejorar el rendimiento de nuestra red neuronal en función de la precisión, por lo que agregamos métricas como precisión.

```
# Compilar red neuronal

classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

Se deben ejecutar varios códigos en este paso.

Ajuste de la red neuronal al conjunto de entrenamiento. Ahora entrenamos nuestro modelo en los datos de entrenamiento. Usamos el método **fit** para adaptarse a nuestro modelo. También optimizamos los pesos para mejorar la eficiencia del modelo. Para esto, tenemos que actualizar los pesos. El **tamaño del lote (Batch Size)** es el número de observaciones después de las cuales actualizamos los pesos. La **época (epoch**) es el número total de iteraciones. Los valores de tamaño de lote y época se eligen por el método de prueba y error.

```
classifier.fit(X_train, y_train, batch_size = 10, epochs = 50)
```

Ya estamos en condiciones de hacer predicciones y evaluar el modelo.

```
# Predecir los resultados del Test set

y_pred = classifier.predict(X_test)

y_pred = (y_pred > 0.5)
```

El resultado de la predicción le dará la probabilidad de que el cliente abandone la empresa. Convertiremos esa probabilidad en binario 0 y 1.

Vamos a predecir sobre una observación:

Geography: Spain

Credit Score: 500

Gender: Female

Age: 40

Tenure: 3

Balance: 50000

Number of Products: 2

Has Credit Card: Yes

Is Active Member: Yes

Todos estos valores se codifican, según vimos antes:

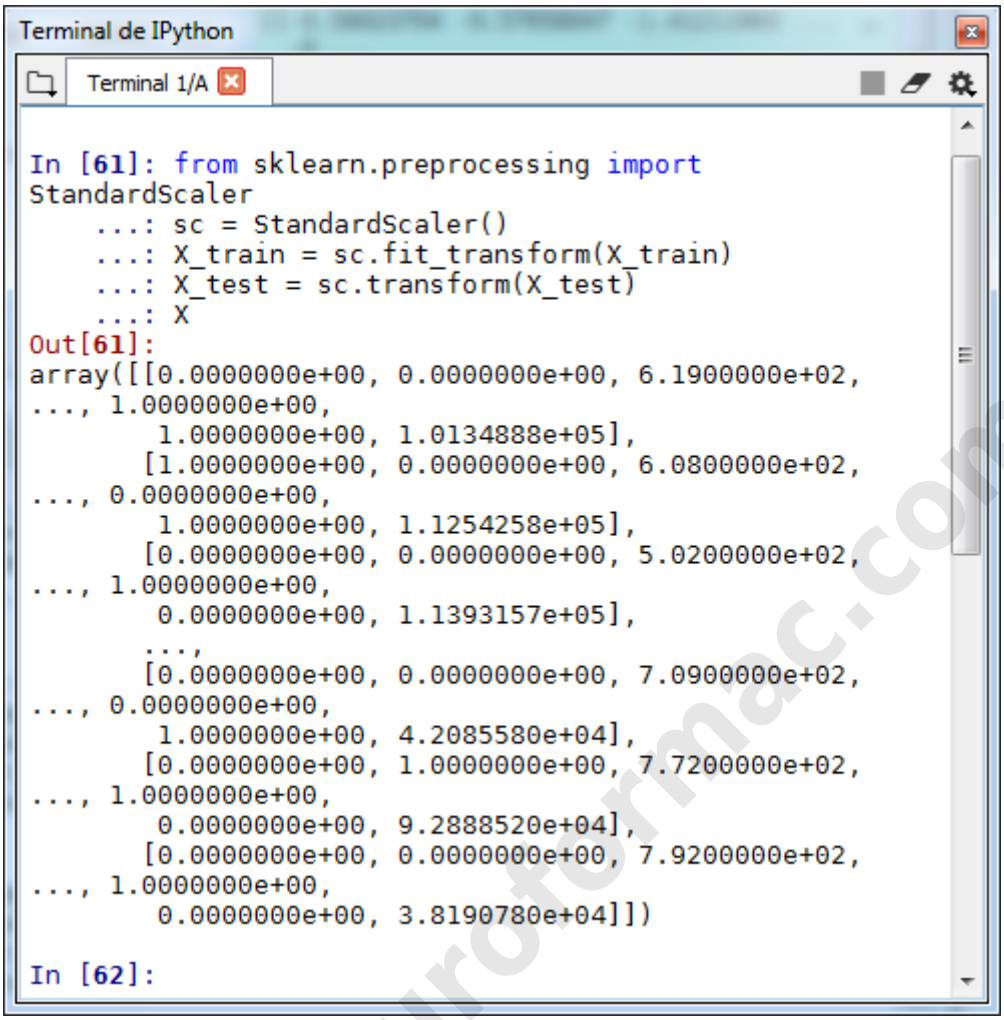
```
new_prediction = classifier.predict(sc.transform
(np.array([[0.0, 0, 500, 1, 40, 3, 50000, 2, 1, 1, 40000]])))
new_prediction = (new_prediction > 0.5)
```

En el último paso evaluamos el rendimiento de **nuestro modelo**. Ya tenemos **resultados** originales y, por lo tanto, podemos construir una matriz de confusión para verificar la precisión de nuestro modelo.

Construimos la **matriz de confusión**:

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print (cm)
```

Después de un tiempo de ejecución, obtenemos lo siguiente:



A partir de la matriz de confusión, la precisión de nuestro modelo se puede calcular como:

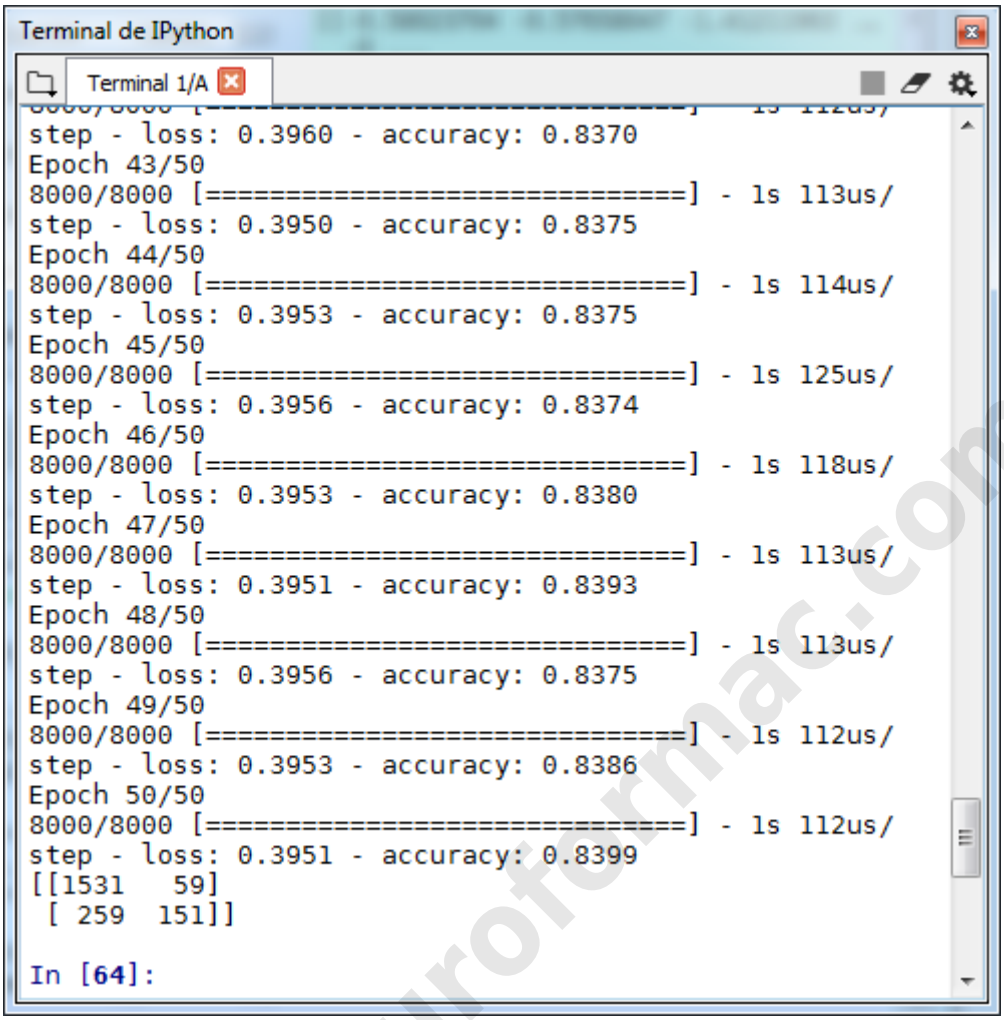
Accuracy = 1531+151/2000=0.841

Logramos una precisión del 84.10%, lo que es un buen resultado.

Podemos ver el resultado de la predicción con:

print(new_prediction)

El resultado es que el cliente no nos abandona.



A continuación, se muestra todo el **código**.

```
# Artificial Neural Network

# Instalar Theano

pip install --upgrade theano

# Instalar Tensorflow

pip install --upgrade tensorflow

# Instalar Keras

pip install --upgrade keras

# Importar las librerias
```

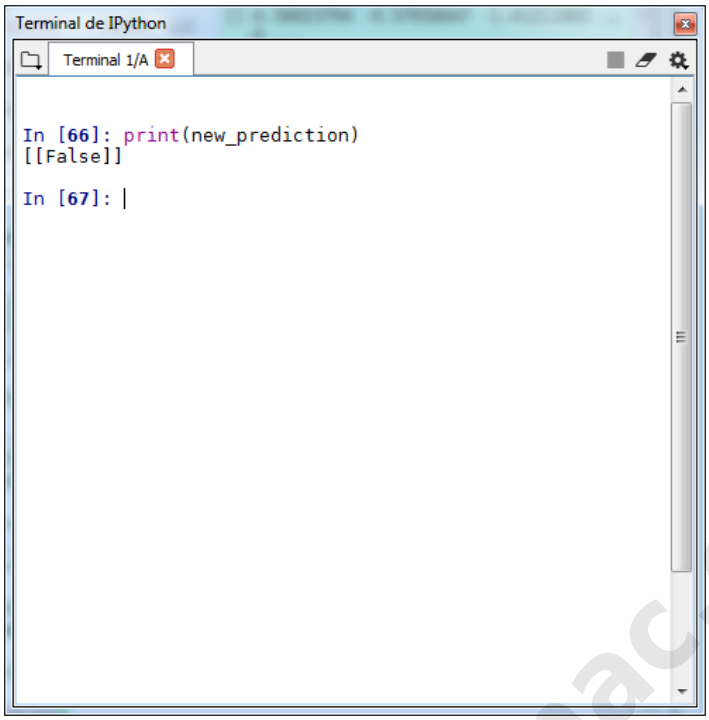
```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importar la base de datos

dataset = pd.read_csv('c:/python/Churn_Modelling.csv')

X = dataset.iloc[:, 3:13].values

Y = dataset.iloc[:, 13].values

X

Y

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

labelencoder_X_1 = LabelEncoder()

X[:,1] = labelencoder_X_1.fit_transform(X[:,1])

labelencoder_X_2 = LabelEncoder()

X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])

X

onehotencoder = OneHotEncoder(ColumnTransformer = [1])

X = onehotencoder.fit_transform(X).toarray()

X = X[:, 1:]

X

#Dividir el dataset en Training set y Test Set

from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.2)

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)

X

# Importar la libreria de Keras y sus paquetes

import keras

from keras.models import Sequential

from keras.layers import Dense

#Inicializar la red neuronal

classifier = Sequential()

classifier.add(Dense(units = 6, kernel_initializer = 'uniform',

activation = 'relu', input_dim = 11))

classifier.add(Dense(units = 6, kernel_initializer = 'uniform',

activation = 'relu'))

classifier.add(Dense(units = 1, kernel_initializer = 'uniform',

activation = 'sigmoid'))

# Compilar red neuronal

classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

```
classifier.fit(X_train, y_train, batch_size = 10, epochs = 50)

# Predecir los resultados del Test set

y_pred = classifier.predict(X_test)

y_pred = (y_pred > 0.5)

new_prediction = classifier.predict(sc.transform

(np.array([[0.0, 0, 500, 1, 40, 3, 50000, 2, 1, 1, 40000]])))

new_prediction = (new_prediction > 0.5)

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

print (cm)

print(new_prediction)
```

4El algoritmo de propagación directa

En esta sección, aprenderemos cómo **escribir código** para hacer propagación hacia adelante **(predicción)** para una red neuronal simple.



En el algoritmo de propagación directa, cada punto de datos es un cliente. La primera entrada es cuántas cuentas tienen, y la segunda entrada es cuántos hijos tienen. El modelo predecirá cuántas transacciones realizará el usuario en el próximo año.

Los datos de entrada se cargan previamente como datos de entrada, y los pesos están en un diccionario llamado **weights**. El array de pesos para el primer nodo en la capa oculta está en pesos ['node_0'], y para el segundo nodo en la capa oculta está en pesos ['node_1'] respectivamente.

Los pesos que se alimentan al nodo de salida están disponibles en weights.

Una función de activación es una función que funciona en cada nodo. Transforma la entrada del nodo en alguna salida.

La **función de activación lineal rectificada (llamada ReLU)** se usa ampliamente en redes de muy alto rendimiento. Esta función toma un solo número como entrada, devuelve 0 si la entrada es negativa y la entrada como salida si la entrada es positiva.

Aquí hay algunos ejemplos:

- relu (4) = 4
- relu (-2) = 0

Completamos la definición de la función relu():

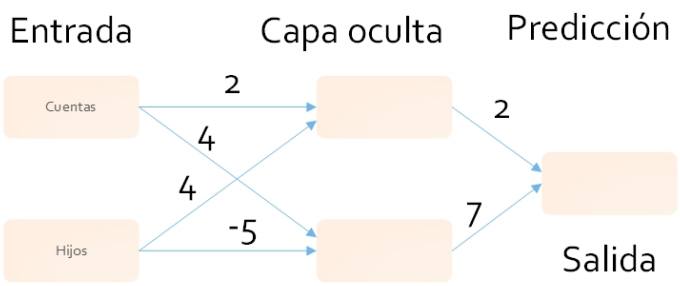
- Usamos la función max() para calcular el valor de la salida de relu().
- Aplicamos la función relu() a node_0_input para calcular node_0_output.
- Aplicamos la función relu() a node_1_input para calcular node_1_output.

Veamos el código:

```
import numpy as np

input_data = np.array([-1, 2])

weights = {

'node_0': np.array([3, 3]),

'node_1': np.array([1, 5]),

'output': np.array([2, -1])

}

node_0_input = (input_data * weights['node_0']).sum()

node_0_output = np.tanh(node_0_input)

node_1_input = (input_data * weights['node_1']).sum()

node_1_output = np.tanh(node_1_input)

hidden_layer_output = np.array(node_0_output, node_1_output)

output =(hidden_layer_output * weights['output']).sum()

print(output)

def relu(input):

'''Definición de relu funcion de activacion'''
```

```
# Calcular el valor de salida de la funcion relu: output

output = max(input,0)

# Retornar el valor recien calculado

return(output)

# Calcular valor de node 0: node_0_output

node_0_input = (input_data * weights['node_0']).sum()

node_0_output = relu(node_0_input)

# Calcular valor de node 1: node_1_output

node_1_input = (input_data * weights['node_1']).sum()

node_1_output = relu(node_1_input)

# Guardar valores en el array: hidden_layer_outputs

hidden_layer_outputs = np.array([node_0_output, node_1_output])

# Calcular salida del modelo (sin aplicar relu)

model_output = (hidden_layer_outputs * weights['output']).sum()

print(model_output)# Mostrar salida del modelo
```

La salida es la siguiente:



A continuación, vamos a desarrollar una función llamada **predic_with_network().** Esta función

generará predicciones para múltiples observaciones de datos, tomadas de la red anterior tomada como input_data. Se están utilizando los pesos dados en la red anterior. La definición de la función relu () también se está utilizando.

Definamos una función llamada predic_with_network () que acepta dos argumentos - input_data_row y weights - y devuelve una predicción de la red como salida.

Calculamos los valores de entrada y salida para cada nodo, almacenándolos como: node_0_input, node_0_output, node_1_input y node_1_output.

Para calcular el **valor de entrada** de un nodo, multiplicamos las matrices relevantes y calculamos su suma.

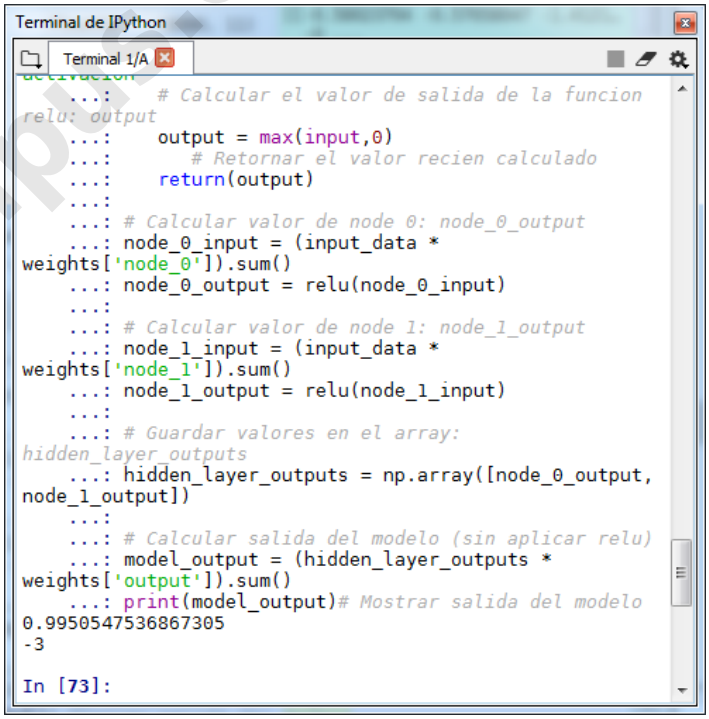
Para calcular el valor de salida de un nodo, aplicamos la función relu () al valor de entrada del nodo. Usamos un 'bucle for' para iterar sobre input_data -

También usamos nuestro predict_with_network () para generar predicciones para cada fila de input_data - input_data_row. Finalmente, agregamos cada predicción a los resultados.

```
# Definir predict_with_network()

def predict_with_network(input_data_row, weights):

# Calcular valor node 0

node_0_input = (input_data_row * weights['node_0']).sum()

node_0_output = relu(node_0_input)

# Calcular valor node 1

node_1_input = (input_data_row * weights['node_1']).sum()

node_1_output = relu(node_1_input)

# Guardar los valores de node en un array: hidden_layer_outputs

hidden_layer_outputs = np.array([node_0_output, node_1_output])
```

```
# Calcular salida del modelo

input_to_final_layer = (hidden_layer_outputs*weights['output']).sum()

model_output = relu(input_to_final_layer)

# Retornar salida del modelo

return(model_output)

# Crear lista vacia para almacenar los resultados de la prediccion

results = []

for input_data_row in input_data:

# Agregar prediccion a los resultados

results.append(predict_with_network(input_data_row, weights))

print(results)# Mostrar resultados
```

La salida es la siguiente:



Aquí hemos usado la función relu donde relu (26) = 26 y relu (-13) = 0 y así sucesivamente.

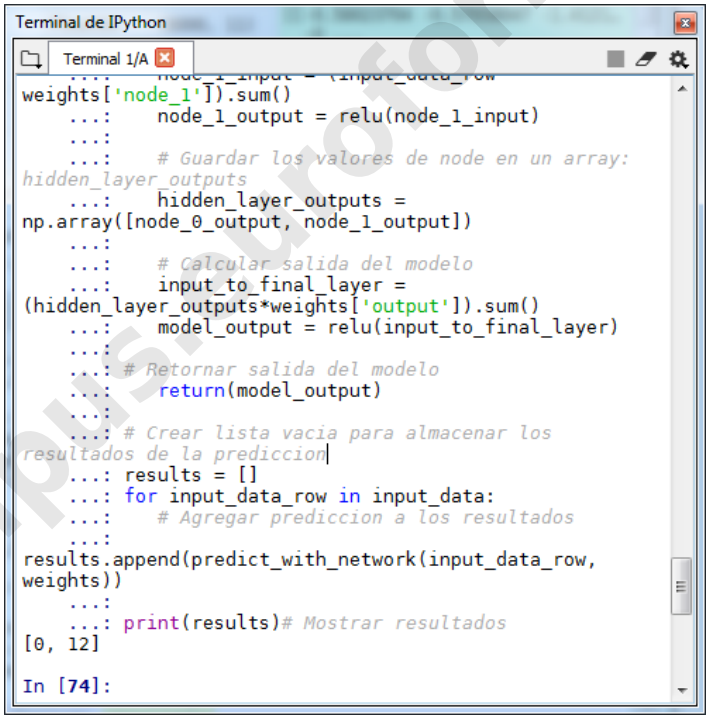
Redes neuronales profundas multicapa

Vamos a desarrollar un código para hacer propagación hacia adelante para una red neuronal con dos capas ocultas. Cada capa oculta tiene dos nodos. Los datos de entrada se han precargado como **input_data**. Los nodos en la primera capa oculta se denominan node_0_0 y node_0_1.

Sus pesos están precargados como weights['node_0_0'] y weights['node_0_1'] respectivamente.

Los nodos en la segunda capa oculta se llaman **nodo_1_0** y **node_1_1**. Sus pesos están precargados como **weights['node_1_0']** y **weights ['node_1_1']** respectivamente.

Luego creamos un modelo de salida de los nodos ocultos usando pesos precargados como **weights['output']**.



Calculamos node_0_0_input usando sus pesos weights ['node_0_0'] y los input_data dados. Luego aplicamos la función relu () para obtener node_0_0_output.

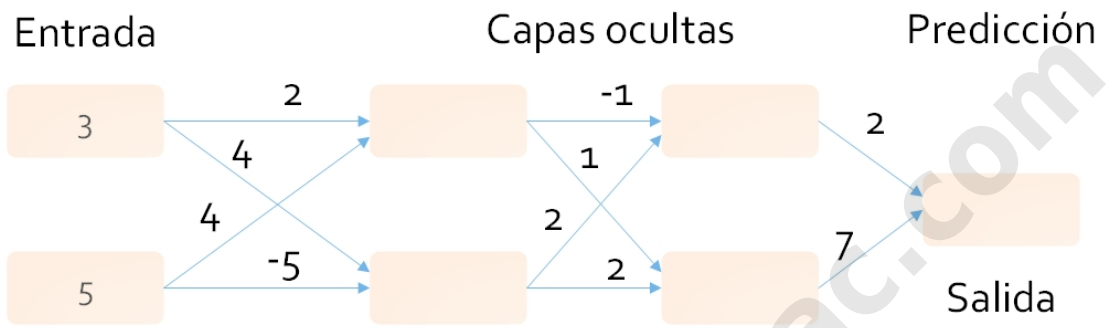
Hacemos lo mismo que antes para node_0_1_input para obtener node_0_1_output.

Calculamos node_1_0_input usando sus pesos weights ['node_1_0'] y las salidas de la primera capa

oculta: hidden_0_outputs. Luego aplicamos la función relu () para obtener node_1_0_output.

Hacemos lo mismo que anteriormente para node_1_1_input para obtener node_1_1_output.

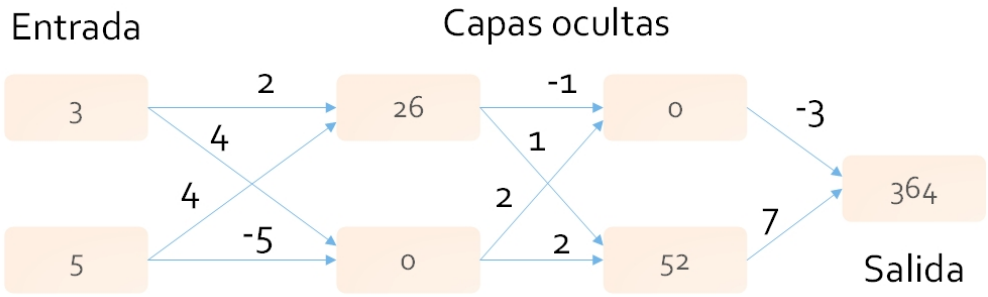
Calculamos model_output usando pesos ['output'] y los resultados de la segunda capa oculta array hidden_1_outputs. No aplicamos la función relu () a esta salida.




```
import numpy as np

input_data = np.array([3, 5])

weights = {

'node_0_0': np.array([2, 4]),

'node_0_1': np.array([4, -5]),

'node_1_0': np.array([-1, 1]),

'node_1_1': np.array([2, 2]),

'output': np.array([2, 7])

}

def predict_with_network(input_data):

# Calcular node 0 en la primera capa oculta

node_0_0_input = (input_data * weights['node_0_0']).sum()
```

```
node_0_0_output = relu(node_0_0_input)

# Calcular node 1 en la primera capa oculta

node_0_1_input = (input_data*weights['node_0_1']).sum()

node_0_1_output = relu(node_0_1_input)

# Guardar los valores del node values en el array: hidden_0_outputs

hidden_0_outputs = np.array([node_0_0_output, node_0_1_output])

# Calcular node 0 en la segunda capa oculta

node_1_0_input = (hidden_0_outputs*weights['node_1_0']).sum()

node_1_0_output = relu(node_1_0_input)

# Calcular node 1 en la segunda capa oculta

node_1_1_input = (hidden_0_outputs*weights['node_1_1']).sum()

node_1_1_output = relu(node_1_1_input)

# Guardar los valores de node en el array: hidden_1_outputs

hidden_1_outputs = np.array([node_1_0_output, node_1_1_output])

# Calcular salida del modelo: model_output

model_output = (hidden_1_outputs*weights['output']).sum()

# Return la salida del modelo

return(model_output)

output = predict_with_network(input_data)

print(output)
```

La salida es la siguiente:



El objetivo del entrenamiento es hacer que el coste del entrenamiento sea lo más pequeño posible en millones de ejemplos de entrenamiento. Para hacer esto, la red ajusta los pesos y los sesgos hasta que la predicción coincida con la salida correcta.

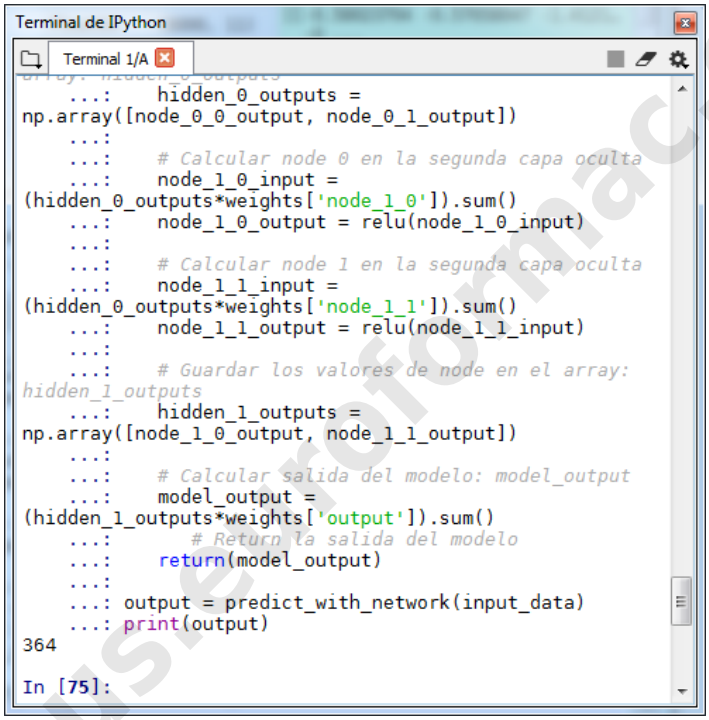
- Verdadero.
- Falso.

En la década de 1960, los SVM se introdujeron por primera vez, y se refinaron en 1990. Los SVM tienen una forma única de implementación en comparación con otros algoritmos de aprendizaje automático. En la actualidad, son muy populares debido a su capacidad para trabajar con múltiples variables continuas y categóricas.

- Verdadero.
- Falso.

Combinación de modelos. Random Forest.

Cada **árbol de decisión** tiene una alta varianza, pero cuando los combinamos todos juntos en paralelo, la varianza resultante es baja, ya que cada árbol de decisión se entrena perfectamente en esos datos de muestra en particular y, por lo tanto, la salida no depende de un árbol de decisión sino de múltiples árboles de decisión. En el caso de un problema de clasificación, el resultado final se obtiene utilizando el clasificador de voto mayoritario. En el caso de un problema de regresión, la salida final es la media de todas las salidas. Esta parte es Agregación.

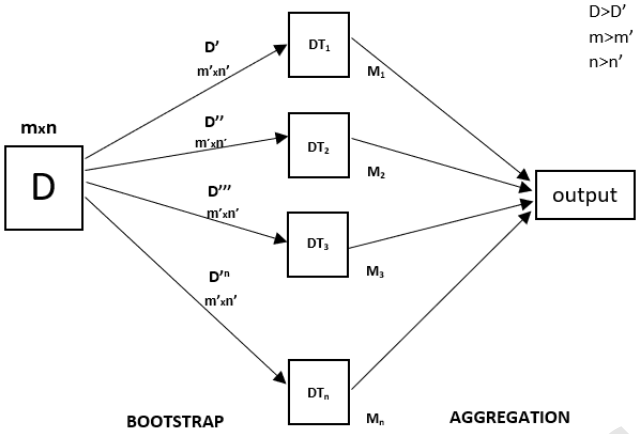


Un **Random Forest** es una técnica de conjunto (ensamblado) capaz de realizar tareas tanto de regresión como de clasificación con el uso de múltiples árboles de decisión y una técnica llamada Bootstrap y Aggregation, comúnmente conocida como embolsado (bagging). La idea básica detrás de esto es combinar múltiples árboles de decisión para determinar el resultado final en lugar de confiar en árboles de decisión individuales.

Random Forest tiene múltiples árboles de decisión como **modelos de aprendizaje base**. Realizamos aleatoriamente el muestreo de filas y el muestreo de características del conjunto de datos que forman conjuntos de datos de muestra para cada modelo. Esta parte se llama Bootstrap.

El bosque aleatorio es un modelo de conjunto que hace crecer múltiples árboles y clasifica objetos

según los "votos" de todos los árboles, es decir, un objeto se asigna a una clase que tiene más votos de todos los árboles. Al hacerlo, el problema con un alto sesgo (sobreajuste) podría aliviarse.



Random Forest es un metaestimulador que se ajusta a varios árboles de decisión en varias submuestras de conjuntos de datos y utiliza el promedio para mejorar la precisión predictiva del modelo y controla el sobreajuste. El tamaño de la submuestra siempre es el mismo que el tamaño de la muestra de entrada original, pero las muestras se extraen con reemplazo.

Características de Random Forest

Las principales ventajas de Random Forest son las siguientes:

- Podría manejar un gran conjunto de datos con alta dimensionalidad, salida en función de la importancia de la variable, útil para explorar los datos
- Podría manejar falta de datos mientras se mantiene la precisión.
- En la mayoría de los casos, la reducción y ajuste del clasificador Random Forest son más precisos que los árboles de decisión.
- Las desventajas de Random Forest son las siguientes:
    - Podría ser una caja negra, los usuarios tienen poco control sobre lo que hace el modelo
    - La predicción es lenta en tiempo real, difícil de implementar y el algoritmo es complejo.
    - Si comparamos Random Forest con Árbol de decisión, encontramos las siguientes diferencias:

- Los bosques aleatorios son un conjunto de múltiples árboles de decisión.
- Los árboles de decisión profunda pueden sufrir de sobreajuste, pero los bosques aleatorios evitan el sobreajuste mediante la creación de árboles en subconjuntos aleatorios.

Los árboles de decisión son computacionalmente más rápidos

Los bosques aleatorios son difíciles de interpretar, mientras que un árbol de decisión es fácilmente interpretable y se puede convertir en reglas.

Implementación de un Random Forest

Vamos a construir un modelo en el conjunto de datos de flores de iris, que es un conjunto de clasificación muy famoso. Comprende la longitud del sépalo, el ancho del sépalo, la longitud del pétalo, el ancho del pétalo y el tipo de flores. Hay tres especies o clases: setosa, versicolor y virginia.

Construiremos un **modelo para clasificar el tipo de flor**. El conjunto de datos está disponible en la biblioteca scikit-learn o también se puede descargar desde el repositorio de aprendizaje automático de UCI.

Comenzamos importando la biblioteca de conjuntos de datos desde scikit-learn y cargamos el conjunto de datos de iris con .load_iris()

```
#Import scikit-learn dataset library

from sklearn import datasets

#Cargar dataset

iris = datasets.load_iris()
```

Podemos imprimir los **nombres de destino y de características** para asegurarnos de que tenemos el conjunto de datos correcto:

```
# mostrar las etiquetas de especies (setosa, versicolor,virginica)

print(iris.target_names)
```

```
# mostrar los nombres de las cuatro características

print(iris.feature_names)
```

El resultado es el esperado:

```
['setosa' 'versicolor' 'virginica']

['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

Vamos a explorar un poco los datos, para ello, mostraremos las primeras cinco filas del conjunto de datos, así como la variable de destino para todo el conjunto de datos.

```
# mostrar datos iris (primeros 5 registros)

print(iris.data[0:5])

# mostrar las etiquetas iris (0:setosa, 1:versicolor, 2:virginica)

print(iris.target)
```

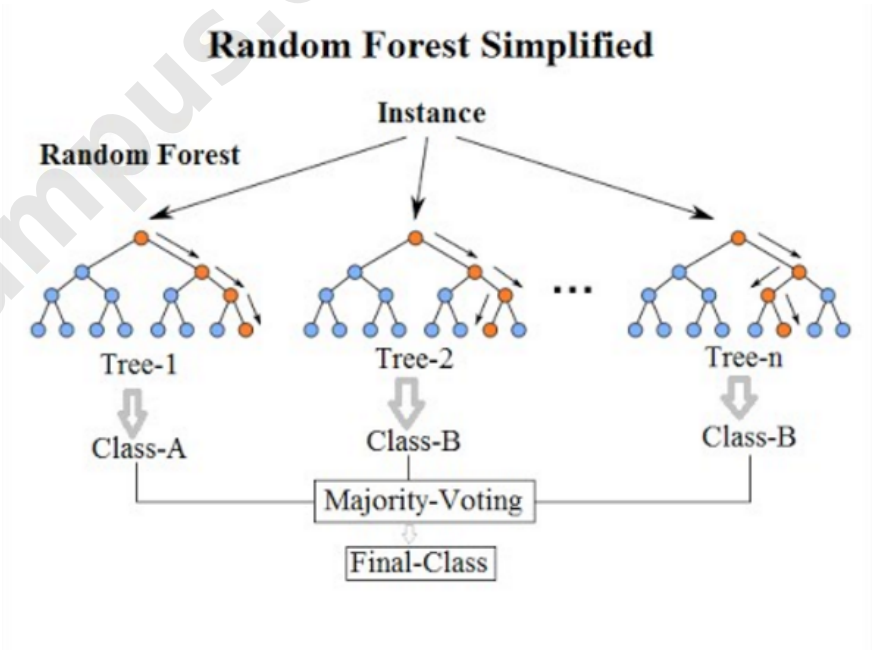
La salida es:

```
[[ 5.1 3.5 1.4 0.2]

[ 4.9 3. 1.4 0.2]

[ 4.7 3.2 1.3 0.2]

[ 4.6 3.1 1.5 0.2]

[ 5. 3.6 1.4 0.2]]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

2 2]

A continuación, vamos a crear un **DataFrame del conjunto de datos de iris**:

```
# Crear un DataFrame a partir del dataset iris.

import pandas as pd

data=pd.DataFrame({

'sepal length':iris.data[:,0],

'sepal width':iris.data[:,1],

'petal length':iris.data[:,2],

'petal width':iris.data[:,3],

'species':iris.target

})

data.head()
```



En primer lugar, se separan las columnas en variables dependientes e independientes (o

características y etiquetas). Luego se dividen esas variables en un conjunto de entrenamiento y prueba.

```
# Import train_test_split function

from sklearn.model_selection import train_test_split

X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features

y=data['species'] # Labels

# Dividir dataset en training set y test set

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training y 30% test
```

Después de dividir, se entrena el modelo con el **conjunto de entrenamiento** y se realizarán predicciones en el conjunto de prueba.

```
#Import Random Forest Model

from sklearn.ensemble import RandomForestClassifier

#Crear Gaussian Classifier

clf=RandomForestClassifier(n_estimators=100)

#Entrenar el model utilizando training sets y_pred=clf.predict(X_test)

clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)
```

Después del entrenamiento, verificamos la precisión utilizando valores reales y predichos.

```
#Importar scikit-learn metrics module para cálculo de exactitud

from sklearn import metrics

# Model Accuracy, ¿Con que frecuencia es correcto el clasificador?
```

```
print("Exactitud:",metrics.accuracy_score(y_test, y_pred))
```

```
('Exactitud:', 0.93333333333333335)
```

También es posible hacer una predicción para un solo elemento, por ejemplo:

Longitud del sépalo = 3

Ancho del sépalo = 5

Longitud del pétalo = 4

Ancho de pétalos = 2

Ahora podemos predecir qué tipo de flor es.

```
clf.predict([[3, 5, 4, 2]])
```

El resultado es:

```
array([2])
```

El valor 2 indica el tipo de flor Virginica.

El código completo es el siguiente:

```
#Import scikit-learn dataset library
from sklearn import datasets
#Cargar dataset
iris = datasets.load_iris()
# mostrar las etiquetas de especies (setosa, versicolor,virginica)
print(iris.target_names)
# mostrar los nombres de las cuatro características
print(iris.feature_names)
```

```
# mostrar datos iris (primeros 5 registros)

print(iris.data[0:5])

# mostrar las etiquetas iris (0:setosa, 1:versicolor, 2:virginica)

print(iris.target)

# Crear un DataFrame a partir del dataset iris.

import pandas as pd

data=pd.DataFrame({

'sepal length':iris.data[:,0],

'sepal width':iris.data[:,1],

'petal length':iris.data[:,2],

'petal width':iris.data[:,3],

'species':iris.target

})

data.head()

# Import train_test_split function

from sklearn.model_selection import train_test_split

X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features

y=data['species'] # Labels

# Dividir dataset en training set y test set

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training y 30% test

#Import Random Forest Model
```

```
from sklearn.ensemble import RandomForestClassifier

#Crear Gaussian Classifier

clf=RandomForestClassifier(n_estimators=100)

#Entrenar el model utilizando training sets y_pred=clf.predict(X_test)

clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

#Importar scikit-learn metrics module para cálculo de exactitud

from sklearn import metrics

# Model Accuracy, ¿Con que frecuencia es correcto el clasificador?

print("Exactitud:",metrics.accuracy_score(y_test, y_pred))

# Predecir tipo de flor

clf.predict([[3, 5, 4, 2]])
```

Con Scikit-learn podemos encontrar **características importantes**. En este caso, estamos encontrando características importantes o seleccionando características en el conjunto de datos IRIS.

En scikit-learn, puede realizar esta tarea en los siguientes pasos:

- En primer lugar, creamos un modelo de bosques aleatorios.
- En segundo lugar, utilizamos la variable de importancia de la característica para ver las puntuaciones de importancia de la característica.

En tercer lugar, visualizar estas **puntuaciones** utilizando la **biblioteca seaborn**.

```
from sklearn.ensemble import RandomForestClassifier

#Crear Gaussian Classifier
```

```
clf=RandomForestClassifier(n_estimators=100)

#Entrenar el modelo usando training sets y_pred=clf.predict(X_test)

clf.fit(X_train,y_train)

RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',

max_depth=None, max_features='auto', max_leaf_nodes=None,

min_impurity_decrease=0.0, min_impurity_split=None,

min_samples_leaf=1, min_samples_split=2,

min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,

oob_score=False, random_state=None, verbose=0,

warm_start=False)

import pandas as pd

feature_imp =
pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(ascending=False)

feature_imp
```

El resultado es:

```
petal width (cm) 0.458607

petal length (cm) 0.413859

sepal length (cm) 0.103600

sepal width (cm) 0.023933

dtype: float64
```

También podemos visualizar la importancia de la característica. Las **visualizaciones** son fáciles de

entender e interpretables.

Para la **visualización**, podemos utilizar una combinación de matplotlib y seaborn. Debido a que seaborn está construido sobre matplotlib, ofrece una serie de temas personalizados y proporciona tipos de trama adicionales. **Matplotlib** es un superconjunto de seaborn y ambos son igualmente de importantes para las buenas visualizaciones.

```
import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline

# Crear un bar plot

sns.barplot(x=feature_imp, y=feature_imp.index)

# Añadir etiquetas al gráfico

plt.xlabel('Feature Importance Score')

plt.ylabel('Features')

plt.title("Visualizing Important Features")

plt.legend()

plt.show()
```

	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Para generar el **modelo en funciones** seleccionadas, podemos por ejemplo eliminar la función de "ancho de sépalo" porque tiene muy poca importancia y seleccionar las 3 características restantes.

```
# Import train_test_split function

from sklearn.model_selection import train_test_split

# Dividir dataset en features y labels

X=data[['petal length', 'petal width','sepal length']] # Removed feature "sepal width"

y=data['species']

# Dividir dataset en training set y test set

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.70, random_state=5) # 70%
training y 30% test
```

Después de la división, se genera un modelo sobre las características seleccionadas del conjunto de entrenamiento, realizará predicciones sobre las características del conjunto de pruebas seleccionadas y comparará los valores reales y previstos.

```
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier

clf=RandomForestClassifier(n_estimators=100)

#Entrenar el modelo usando training sets y_pred=clf.predict(X_test)

clf.fit(X_train,y_train)

# predicción sobre test set

y_pred=clf.predict(X_test)

#Importar scikit-learn metrics module para cálculo de exactitud

from sklearn import metrics

# Model Accuracy, ¿Con que frecuencia es correcto el clasificador?

print("Exactitud:",metrics.accuracy_score(y_test, y_pred))
```

(Exactitud:', 0.95238095238095233)

Podemos ver que después de eliminar las **características** menos importantes **(anchura del sépalo)**, la precisión aumentó. Esto se debe a que se eliminaron datos engañosos y ruido, lo que resultó en una mayor precisión. Una menor cantidad de características también reduce el tiempo de entrenamiento.

Relaciona los elementos de la columna Derecha con la columna Izquierda

El bosque aleatorio	1	[ ] Es una técnica de conjunto (ensamblado) capaz de realizar tareas tanto de regresión como de clasificación con el uso de múltiples árboles de decisión y una técnica llamada Bootstrap y Aggregation, comúnmente conocida como embolsado (bagging).
Un Random Forest	2	[ ] Es un modelo de conjunto que hace crecer múltiples árboles y clasifica objetos según los "votos" de todos los árboles, es decir, un objeto se asigna a una clase que tiene más votos de todos los árboles.

Recuerda

[[[Elemento Multimedia]]]

Autoevaluación

¿Cómo “aprende” una red neuronal?

- Mediante programación.
- A través de ejemplos.
- Utilizando una base de datos.

¿Cuál es la salida de una red perceptrón de una sola capa?

- Suma ponderada de todas sus entradas.
- Suma ponderada de todas sus capas ocultas.
- Función sigmoide de las capas ocultas.

En un modelo, ¿Qué porcentaje de datos se suele asignar a entrenamiento?

- 100%
- 70%
- 30%

¿Cuál de los siguientes es un modelo lineal?

SVM.

Redes neuronales.

Regresión Logística.

¿Cuál de las siguientes es una ventaja de los árboles de decisión?

Generaliza bien.

Pequeñas variaciones en los datos generan un árbol completamente diferente.

Fácil de visualizar.