ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

ESTUDIO COMPARATIVO DE LOS ALGORITMOS DE CLASIFICACIÓN SUPERVISADA EMPLEANDO DATOS ARTIFICIALES

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES

EDDY PATRICIO LLUMIQUINGA ALMEIDA

DIRECTOR: PhD. ROBIN GERARDO ÁLVAREZ RUEDA

Quito, Febrero 2022

AVAL

Certifico que el presente trabajo fue desarrollado por Eddy Patricio Llumiquinga Almeida, bajo mi supervisión.

Ph.D. Robin Álvarez Rueda

DIRECTOR DEL TRABAJO DE TITULACIÓN

DECLARACIÓN DE AUTORÍA

Yo, Eddy Patricio Llumiquinga Almeida, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración dejo constancia de que la Escuela Politécnica Nacional podrá hacer uso del presente trabajo según los términos estipulados en la Ley, Reglamentos y Normas vigentes.

Eddy Patricio Llumiquinga Almeida

DEDICATORIA

Dedico el proyecto a mis padres, como muestra de gratitud y admiración, por ser ejemplo de lucha, amor y por formar al hombre en el que me he convertido.

A mis hermanos, por su adorable compañía y apoyo.

A mi novia, quien me ha dado mucho más que una palabra de aliento ha sido refugio en épocas dificultosas, que apenas está comenzando.

A Dios, que me dio la dicha de formar parte de sus vidas.

AGRADECIMIENTO

Quiero expresar mi gratitud a Dios, quien con su gracia ha hecho posible la culminación de este trabajo de investigación.

A mis padres, Elena y Edgar, que me han dado su apoyo incondicional, en las buenas y en las malas.

A mis hermanos, que me han animado a seguir adelante sin importar las circunstancias.

A mi novia Estefany, por todo su apoyo, no perder la fe en mí y ayudarme a ser una mejor persona constantemente.

A mi tutor, el PhD Robin Álvarez, quien sin importar la hora brindó su apoyo y conocimiento de manera desinteresada. Sus reparos acertados permitieron el desarrollo del presente proyecto.

A mis profesores, que con su sabiduría han hecho de mí una gran profesional.

ÍNDICE DE CONTENIDO

RESUMEN

El aprendizaje de máquina (machine learning) es un área de la inteligencia artificial que está en auge y hoy en día se utiliza en todo tipo de aplicaciones. Si bien existen investigaciones en campos específicos en los que se comparan algunos métodos de clasificación supervisada para ver cuál de ellos es el mejor, ninguno de ellos realiza un análisis comparativo que permita determinar de manera general cuál de estos métodos es el mejor y la respuesta a esta pregunta siempre está pendiente.

En este trabajo se revisa la parte teórica y se realiza su implementación en Matlab de algunos de los algoritmos de clasificación supervisada más utilizados: MLP (MultiLayer Perceptron o Perceptrón Multicapa), PNN (Probabilistic Neural Network o Red Neuronal Probabilística), K-NN (K -Near Neighbor o K vecinos más cercanos), NB (Naive Bayes), SVM (Support Vector Machines o Máquina de soporte vectorial) y DT (Decision Tree o Árboles de Decisión). La principal estrategia introducida en este trabajo es la utilización de datos artificiales, ubicados en un espacio bidimensional, que están clasificados bajo una frontera de decisión no lineal también generada artificialmente. Esto permite tener bajo control los siguientes parámetros: la cantidad de datos de entrenamiento asignadas a las distintas clases, la cantidad de datos de prueba asignadas a las distintas clases, el grado de asimetría entre estos (puede haber más datos de entrenamiento y/o prueba para una de las clases) y su grado de alejamiento a la frontera de decisión (se puede controlar si los datos están cercanos, medianamente alejados y alejados). Este punto de vista sobre la creación de datos artificiales es totalmente novedoso y se constituye en un aporte al estado del arte que permite realmente comparar los distintos métodos de clasificación supervisada de modo que ahora sí se pueden emitir conclusiones firmes respecto de la exactitud obtenida por ellos.

PALABRAS CLAVE: algoritmos de clasificación, generación de datos artificiales, comparación de algoritmos de clasificación supervisada.

ABSTRACT

Machine learning is an area of artificial intelligence that is booming and today is used in all kinds of applications. Although there is research in specific fields in which some supervised classification methods are compared to see which of them is the best, none of them performs a comparative analysis to determine in a general way which of these methods is the best and the answer to this question is always pending.

This work reviews the theoretical part and performs its implementation in Matlab of some of the most used supervised classification algorithms: MLP (MultiLayer Perceptron), PNN (Probabilistic Neural Network), K-NN (K -Near Neighbor), NB (Naive Bayes), SVM (Support Vector Machines) and DT (Decision Tree). The main strategy introduced in this work is the use of artificial data, located in a two-dimensional space, which are classified under a nonlinear decision boundary also artificially generated. This allows to have under control the following parameters: the amount of training data assigned to the different classes, the amount of test data assigned to the different classes, the degree of asymmetry between these (there can be more training and/or test data for one of the classes) and their degree of remoteness to the decision boundary (it can be controlled whether the data are close, moderately remote, and far away). This point of view on the creation of artificial data is totally novel and constitutes a contribution to the state of the art that really allows comparing the different supervised classification methods so that firm conclusions can now be drawn regarding the accuracy obtained by them.

KEYWORDS: classification algorithms, artificial data generation, comparison of supervised classification algorithms.

1. INTRODUCCIÓN

El aprendizaje automático (en inglés machine learning) es un área de la inteligencia artificial que está en auge en los últimos años en todo tipo de aplicaciones y con mayor razón en las situaciones en donde es muy complejo o imposible de obtener un modelo físico/matemático del fenómeno.

Cuando el aprendizaje automático consiste en crear un algoritmo que sea capaz de aprender el comportamiento específico de unos datos en situaciones conocidas, es decir, si pertenecen bien a una clase o escenario o bien a otro, denominando a estos datos como conjunto de entrenamiento, dicho algoritmo cae dentro de los métodos denominados de clasificación supervisada. Una vez aprendido dicho comportamiento, aquel algoritmo es puesto a prueba por medio de un nuevo conjunto de datos denominado conjunto de prueba, de aquí se obtendrá la bondad del clasificador empleado. Las aplicaciones están en todas las áreas, por ejemplo: motores de búsqueda, diagnósticos médicos, análisis del mercado de valores, juegos, reconocimiento del habla y del lenguaje escrito, etc.[1].

Como sucede en varios campos de la ciencia, para el aprendizaje automático se han desarrollado varios tipos de algoritmos entre los cuales están: MLP (Multilayer Perceptron), PNN (Probabilistic Neural Network), K-NN (K-Near Neighbor), NB (Naive Bayes), SVM (Support Vector Machines), DT (Decision Tree), etc. Desafortunadamente, si bien existen investigaciones en campos específicos en los que se comparan dichos métodos para ver cuál de ellos es el mejor, no existe un estudio comparativo empleando datos generados artificialmente y que permitan evaluar dichos métodos. Por ejemplo, en [2] se hace una comparación de los métodos K-NN, NB y SVM para la detección de línea de corte en sistemas de protección eléctrica, siendo el clasificador NB. En [3], se compara algunos métodos de clasificación supervisada para la detección de la enfermedad de Parkinson en donde se concluye que SVM es el clasificador más efectivo ya que presenta la mejor sensibilidad, especificidad y exactitud para este caso.

En [4], se comparan 6 algoritmos de aprendizaje automático supervisados para clasificar el tráfico de red en donde se concluye que el algoritmo Random Forest tiene mejores resultados en cuanto se refiere a exactitud y precisión.

A partir del análisis del estado del arte se puede concluir que los resultados sobre el mejor método de clasificación supervisada cambian según el tipo de datos empleados y no se ha encontrado un trabajo que generalice dicha comparación. Existen varias características de los algoritmos que permiten decidir qué tan bueno es un método. Para encontrar dichas características se puede plantear las siguientes interrogantes:

a) ¿El algoritmo permite conocer el grado de pertenencia a la clase seleccionada o solamente dice la clase a la que pertenece?

No es lo mismo que un dato esté muy cercano a la frontera de decisión o que esté muy alejado.

b) ¿La cantidad de datos de entrenamiento influye en la exactitud del método?

En este caso, se evaluará al algoritmo, en función de la cantidad de datos de entrenamiento que requiere para tener una exactitud aceptable, no es lo mismo que se requieran miles de datos a que se requieran solo decenas de datos.

c) Si la cantidad de datos de entrenamiento es distinta para las diferentes clases. ¿Influye en la exactitud obtenida por el método?

En aplicaciones reales no se tienen la misma cantidad de datos de entrenamiento para las distintas clases, es por lo que se evalúan los algoritmos cuando la cantidad de muestras de entrenamiento de cada clase son diferentes.

Para una visualización didáctica de la exactitud de un algoritmo de clasificación se tiene una herramienta llamada matriz de confusión, esta tiene dos dimensiones donde las filas de la matriz indican la clase observada o clase real y las columnas indican la clase predicha. Sus beneficios consisten en una visualización sencilla e intuitiva de los parámetros antes mencionados.

Los datos sintéticos o datos artificiales son datos creados según los parámetros de cada usuario para que estos se asemejen al mundo real. Estos además son importantes cuando los datos reales son muy costosos de conseguir o son de difícil acceso. En este caso, como frontera de decisión se tendrá un polinomio rotado y basándose en esta se podrá controlar el grado de cercanía a los datos.

1.1. OBJETIVOS

El objetivo general de este trabajo es estudiar, implementar y comparar los algoritmos más importantes de clasificación supervisada, empleando datos artificiales.

Los objetivos específicos son los siguientes:

- Analizar el estado del arte respecto de los algoritmos de clasificación supervisada.
- Analizar el fundamento teórico de algunos de los algoritmos de clasificación más utilizados en la actualidad.

- Implementar un algoritmo que permita la generación de datos artificiales pertenecientes a las distintas clases.
- Implementar códigos propios de los algoritmos de clasificación antes mencionados.
- Comparar los métodos de clasificación y determinar cuál de ellos obtiene mejores resultados.

1.2. ALCANCE

En este proyecto se generan de forma automática tanto la o las fronteras de decisión no lineales como la cantidad de datos pertenecientes a las distintas clases. Se trabajará máximo con 4 clases.

Como se mencionó anteriormente uno de los principales aportes de este trabajo es la generación automática de una frontera de decisión NO LINEAL (un polinomio determinado) y el conjunto de entrenamiento, de modo que el investigador pueda generar N1 datos pertenecientes a una clase y N2 datos pertenecientes a otra clase y además que pueda decidir el grado de cercanía a la frontera de decisión no lineal. Por ejemplo, en la Figura 1.1 Conjuntos de datos separados en 2 clases, cercanos a la frontera de decisión. se muestran dos conjuntos de datos pertenecientes a dos clases separadas por una frontera de decisión polinomial x^3-4x^2+x-5 y que están muy cercanos a dicha frontera, mientras que en la Figura 1.2 Conjuntos de datos separados en 2 clases relativamente alejados de la frontera de decisión. se muestran dos conjuntos similares a los anteriores con la diferencia de que están más alejados de la frontera.

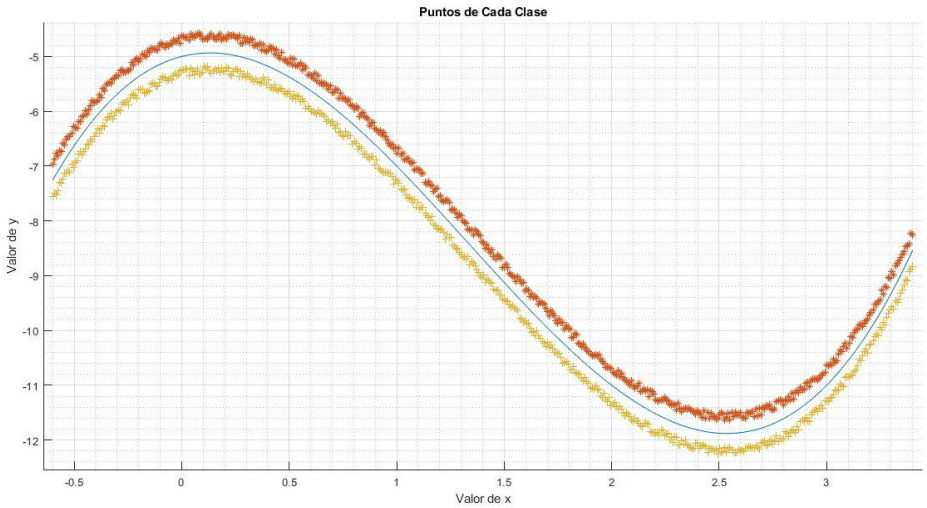


Figura 1.1 Conjuntos de datos separados en 2 clases, cercanos a la frontera de decisión.

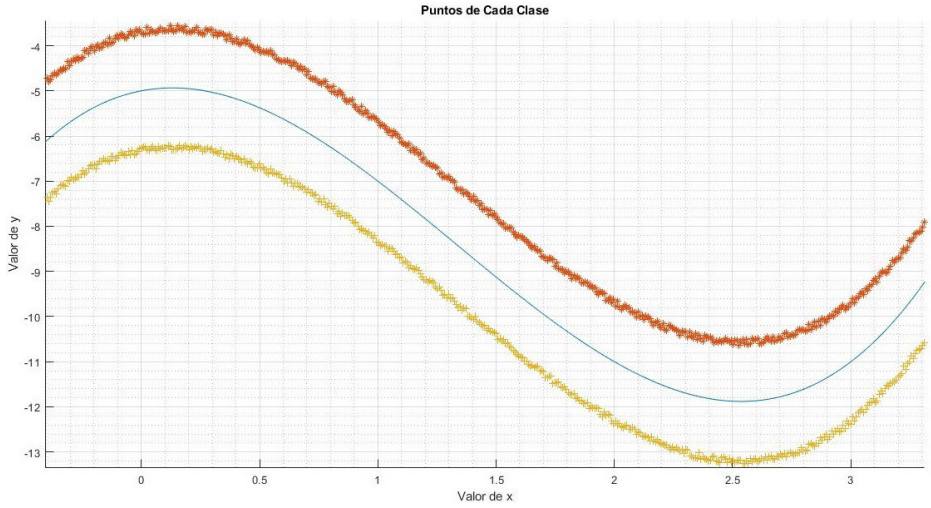


Figura 1.2 Conjuntos de datos separados en 2 clases relativamente alejados de la frontera de decisión.

Por otro lado, se analiza la teoría subyacente y se realiza su implementación de, al menos, los siguientes algoritmos supervisados de clasificación:

- Multi-Layer Perceptron
- Support Machine vector
- Decision Tree
- Naïve Bayes
- K-Near Neighbors
- Probabilistic Neural Network

Para tener una base de partida, se explorarán las funciones incluidas ya en Matlab y posteriormente se procederá a analizar la base teórica de las técnicas antes mencionadas y su implementación, creando funciones propias.

Una vez implementadas nuestras propias funciones, cada uno de los métodos son entrenados empleando el conjunto de entrenamiento generado automáticamente. Respecto de los resultados obtenidos, se analizan algunas características de los métodos de clasificación, por ejemplo, si sólo entrega una decisión respecto de la clase a la que pertenece o si también entregan el grado de pertenencia; si influye o no la cantidad de datos de entrenamiento en el grado de aciertos, si influye o no la cantidad de datos N1 y N2, si estos deben ser iguales o no, etc.

1.3. MARCO TEÓRICO

1.3.1. ESTADO DEL ARTE

A continuación, se revisan algunas investigaciones realizadas en los últimos años con respecto a la comparación de algunos algoritmos de aprendizaje supervisado en distintos

En [3], se comparan los algoritmos de K-NN, NB y SVM para determinar qué algoritmo funciona mejor al detectar enfermedad de Parkinson.

Para comparar los algoritmos se tienen 3 métricas como: la precisión, la exactitud, o sensibilidad. Los resultados indicaron que SVM tiene los mejore resultados (sensibilidad del 77%, precisión del 80% y especificidad del 83%), seguido por K-NN (sensibilidad del 68.2%, precisión del 70% y especificidad del 72%), finalmente NB (sensibilidad del 63.6%, precisión del 65% y especificidad del 66.6%).[3] Se observa en las métricas anteriormente observadas, se concluye que SVM es el mejor clasificador ya que presenta porcentajes más altos en las métricas anteriormente mencionadas.

En [2] se comparan 3 métodos de clasificación (SVM, NB Y K-NN) respecto de la detección de cortes de línea. Como resultado final se determina que NB fue el mejor ya que obtuvo una exactitud del 100%.

En [5] se presenta la comparación de diferentes técnicas de clasificación usando WEKA con el fin de investigar el CCI (Instancias Clasificadas correctamente) y el ICI (Instancias clasificadas incorrectamente) de un grupo de algoritmos de clasificación. Dentro de las técnicas que se utilizan están las siguientes ANN, SVM, NB, FPT y DT, en donde se presentan sus inconvenientes y sus ventajas. De las métricas anteriormente mencionadas se tiene que SVM tiene la más alta CCI del 85.76% y una ICI del 14.23% frente a los demás clasificadores, por lo tanto, se tiene que el mejor algoritmo de clasificación el aprendizaje automático es el clasificador SVM para uso del campo médico bioinformático.

En [6] se presenta una comparación de clasificadores supervisados y características de imagen para la segmentación de pilas de cultivos imágenes las que son capturadas desde un UAV. Entre los algoritmos de clasificación que se estudian son: SVM, RF, MC, k-NN; Finalmente se obtiene qué SVM supera a las demás técnicas clasificación con un puntaje F1-score 88% y un IOU (Intersection-over-Union) del 78.8% lo que indica que los métodos clásicos de segmentación de imágenes basados en texturas de color pueden considerarse una opción para el campo de aplicación incluso con tendencias modernas cómo aprendizaje automático.

En [7] se comparan varias técnicas de clasificación de aprendizaje automático supervisado para determinar qué algoritmo es más eficiente para detectar la diabetes. Se consideran kit algoritmos diferentes tales como: RF, NB, SVM, redes neuronales, DT, JRip y Tabla de decisiones. Los resultados muestran que para el conjunto de datos con 384 muestras se han clasificado correctamente 72.92% de estas, e incorrectamente el 27.08% con el algoritmo SVM, el cual tiene la mayor precisión, mientras que para el conjunto de datos de 768 muestras se tiene que este mismo clasificador tiene el mayor porcentaje de muestras clasificadas correctamente con el 74.34. Los resultados muestran qué SVM Es el algoritmo con el porcentaje de calificación correctamente más alto.

En [4] se comparan el rendimiento de 6 algoritmos de aprendizaje automático supervisados para clasificar en tráfico de red, ya que es un requisito básico para gestionar la calidad de servicio en las redes de comunicaciones. Los resultados con el conjunto total de datos muestras DT tiene 95.4% de precisión y un recall de 95.5%, mientras que el RF tiene un 96.8% de precisión y un 96.8% en recall. Finalmente, los resultados comparativos muestran que los algoritmos como Random Forest and Decision Tree son clasificadores prometedores para el tráfico de red en términos de precisión de clasificación y eficiencia computacional.

1.3.2. MÉTODOS DE CLASIFICACIÓN

1.3.2.1. Multilayer Perceptron (MLP)

Las MLP, son un tipo de redes neuronales artificiales ampliamente utilizadas en el campo del aprendizaje automático. Se basan en modelos estadísticos no lineales que muestran una relación compleja entre entradas y salidas para descubrir un nuevo patrón, estos realizan una gran variedad de tareas como reconocimiento de imágenes, reconocimiento de voz, traducción automática, etc.

A continuación, se verá la teoría subyacente y un ejemplo en Matlab con los datos artificiales que se crearon anteriormente.

1.3.2.1.1. Neuronas artificiales

Para comprender cómo funciona un Perceptrón multicapa primero se debe estudiar a su elemento más simple la cual consiste en una neurona artificial (Figura 1.3). Esta es el elemento de procesamiento básico en el que una salida se calcula multiplicando sus entradas por un vector peso y sumando sus resultados para finalmente aplicar una función de activación a esta suma.

Su representación de forma matemática viene dada por a la siguiente fórmula:

$$y = f\left(\sum_{k=1}^{n} X_k \omega_k + b\right) \quad (1.1)$$

Donde:

y: representa a las salidas de la neurona.

X: al vector de entrada de la neurona.

ω: representa a los pesos de la neurona.

b: representa al bias.

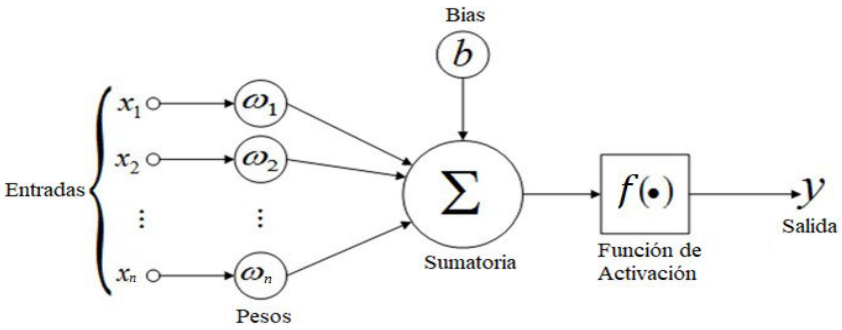


Figura 1.3 Representación de una neurona artificial [8]

Pesos y Bias

Los pesos son valores que en un inicio son aleatorios, pero, a medida que se va entrenado la red, estos van tomando valores hasta que se tiene el dato esperado.

Funciones de Activación

Como se observa en la Figura 1.3 antes de la salida se encuentra una función de activación, también llamada función de activación, esta es fundamental para lograr una buena clasificación, se tiene los siguientes tipos:

A. Función sigmoide o logística.

Es una de las funciones más utilizadas para modelamiento con redes neuronales, tiene un degradado suave, evitando saltos en los valores de salida como se puede apreciar en la Figura 1.4 , matemáticamente se puede expresar de la siguiente manera:

$$f(u) = \frac{1}{1 + e^{-u}} \quad (1.2)$$

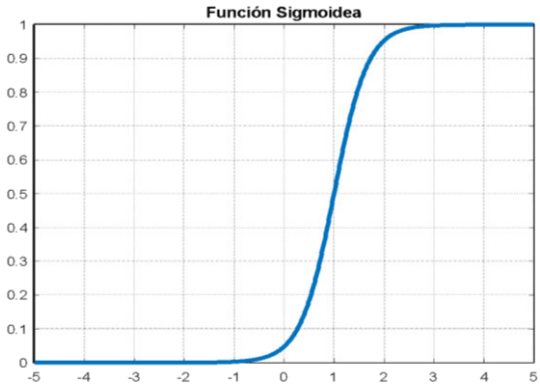


Figura 1.4 Función sigmoidea

Salida de la neurona

La salida de una neurona es básicamente el resultado de la función de activación, esta puede ser un vector o un resultado.

$$y = f(\omega * x + b) \quad (1.3)$$

1.3.2.1.2. Perceptrón multicapa (MLP)

El perceptrón multicapa surge ante la necesidad de clasificar datos cuando no se tiene una frontera de decisión lineal.

Se ha demostrado que el perceptrón multicapa con una capa oculta y la función de activación sigmoidal, puede aproximar cualquier función continua con cualquier grado de precisión deseado, por lo que la MLP se ha denominado como un aproximador universal.

Para dar una aproximación universal, la capa oculta suele ser una neurona con una función de activación sigmoidal, además se debe tener en cuenta que una capa oculta rara vez se usa 2 transformaciones lineales.

$$h = \mathrm{f}(W_1 * x) \quad (1.4)$$

$$y = f(W_2 * h) \quad (1.5)$$

Donde W_1 y W_2son matrices que transforman el vector xa h y esta última a y, las cuales se pueden presentar como una transformación lineal (Figura 1.5).

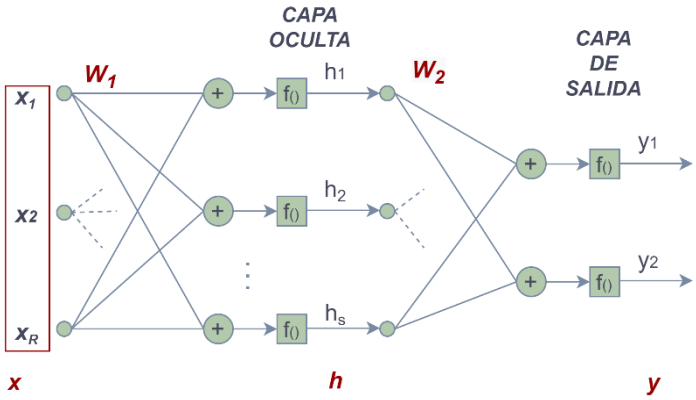


Figura 1.5 Red neuronal con perceptrones en una capa oculta. [9]

1.3.2.1.3. Algoritmo backpropagation

Aplicación del algoritmo Backpropagation para MLP

El algoritmo de Backpropagation toma su nombre debido a que este se lleva a cabo desde la salida hacia la entrada, este se divide en dos partes: debido al ajuste de los pesos de la salida y ajuste de los pesos de la capa oculta.

Por las características mencionadas anteriormente se puede considerar una sola neurona de salida y realizar el análisis como si estuviese aislada, como se observa en la Figura 1.6.

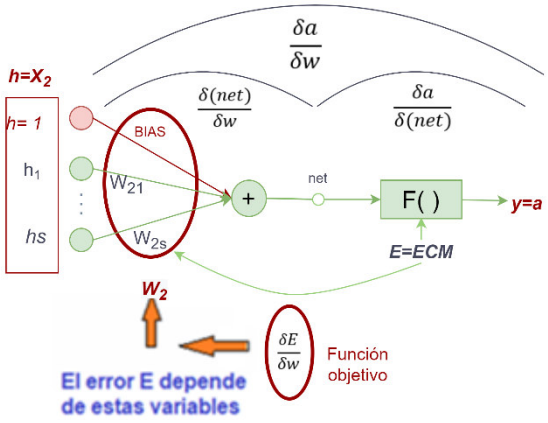


Figura 1.6 Representación de la última capa usando el algoritmo backpropagation. [9]

Al medir el error de las salidas esperadas con respecto a las predichas usamos el Error Cuadrático Medio, esta va a ser la función objetivo para evaluar el algoritmo, se expresa matemáticamente como:

$$\mathrm{E} = \sum_{i=1}^{n} (\mathrm{t}_i - y_i)^2 \qquad (1.6)$$

El primer término es simplemente la variación del error con el valor de salida a. Si t es un vector de n predicciones y y es el vector de los valores verdaderos (salidas de la red).

Lo que se desea es encontrar cómo varía el error con las variaciones de los pesos de la capa de salida. Empleando la regla de la cadena se tiene:

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial a} \cdot \frac{\partial a}{\partial W_2} \quad (1.7)$$

Entonces, al derivar Ecuación (1.6) con respecto a "a":

$$\frac{\delta E}{\delta a} = -2 \cdot (t - a) \quad (1.8)$$

El segundo término de la Ecuación (1.7) podemos dividirlo en otros 2 (Figura 1.6).

$$\frac{\partial a}{\partial W_2} = \frac{\partial a}{\partial (net)} . \frac{\partial (net)}{\partial W_2} \quad (1.9)$$

Y teniendo en cuenta que:

$$net = \sum W_2 . X_2 \quad (1.10)$$

Entonces:

$$\frac{\partial (net)}{\partial W_2} = \frac{\partial \sum W_2 . X_2}{\partial W_2} = X_2 \quad (1.11)$$

$$\frac{\partial a}{\partial (net)} = \frac{\partial f(net)}{\partial (net)} = f'(net) \quad (1.12)$$

De lo que se concluye que f ha de ser derivable.

Si la función **f** de activación es sigmoidal:

$$f(x) = \frac{1}{1 + e^{-x}}; \quad f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f \cdot (1 - f) \quad (1.13)$$

La actualización de los pesos tendrá la variación contraria al error:

$$\Delta w = w(k) - w(k-1) = \frac{\delta E}{\delta W_2} = 2 \cdot (t - a) \cdot f' \cdot X_2 \quad (1.14)$$

Si $f(x) = \mathrm{a}$, se tendría que:

$$\Delta w = 2 \cdot (t - a) \cdot a \cdot (1 - a) \cdot X_2 \quad (1.15)$$

Entonces, la actualización de pesos estaría dada por la siguiente expresión:

$$w(k) = w(k-1) + 2 \cdot (t-a) \cdot a \cdot (1-a) \cdot X_2 \qquad (1.16)$$

¿Qué significado tiene el tamaño del peso?

Ya que el Error (E) es una función que depende de todos los pesos y bias. (Figura 1.6), estos actúan como variables y entonces dicha función de error tendrá una gráfica n-dimensional. Para el caso de tener solo 2 variables (por ejemplo, w_{11} y bias), se tendría una gráfica tridimensional como la mostrada en la Figura 1.7. Si el problema es separable, al ir actualizando los pesos, el error debe ir disminuyendo hasta llegar a un mínimo. Por lo anterior, el tamaño de paso se constituirá en una tasa de aprendizaje (Learning rate o Lr).

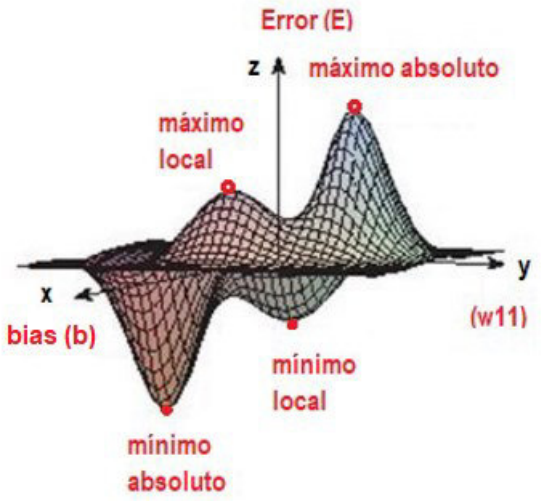


Figura 1.7 Variable de Error (E) si dependiera solamente de dos variables, por ejemplo, w_{11} y el bias (b). [10]

Por lo anterior, se puede intuir fácilmente que este tamaño de paso podría ser muy pequeño con lo que la convergencia sería muy lenta, por el contrario, si fuera muy grande, la convergencia sería muy rápida, pero podría pasarse del mínimo de la función de error. La solución sería emplear una tasa de aprendizaje adaptativa empleando la siguiente estrategia:

ESTRATEGIA:

Si el error disminuyó, entonces incremente la tasa de aprendizaje (agrande el tamaño de paso):

$$Lr = Lr * 1.1$$

Si el error incrementó, entonces disminuye la tasa de aprendizaje (disminuye el tamaño de paso):

$$Lr = Lr * 0.5$$

Solo se actualiza los pesos si el error disminuyó.

Añadiendo una tasa de aprendizaje Lr quedaría:

$$w(k+1) = w(k) + 2 \cdot Lr \cdot (t-a) \cdot a \cdot (1-a) \cdot X_2 \quad (1.17)$$

OBSERVACIÓN.

Desafortunadamente, como se ve en la Figura 1.7, dicha función de error no solo tiene un mínimo global o absoluto, sino que puede tener también mínimos locales y la convergencia podría ocurrir respecto de un mínimo local en lugar de alcanzar el mínimo global, que es lo que realmente interesaría, pero este ya es otro problema.

1.3.2.1.4. Ajuste de pesos de capa oculta

En este caso, la **función de error** debe ser minimizada respecto a los **pesos de la capa oculta (** Figura 1.8).

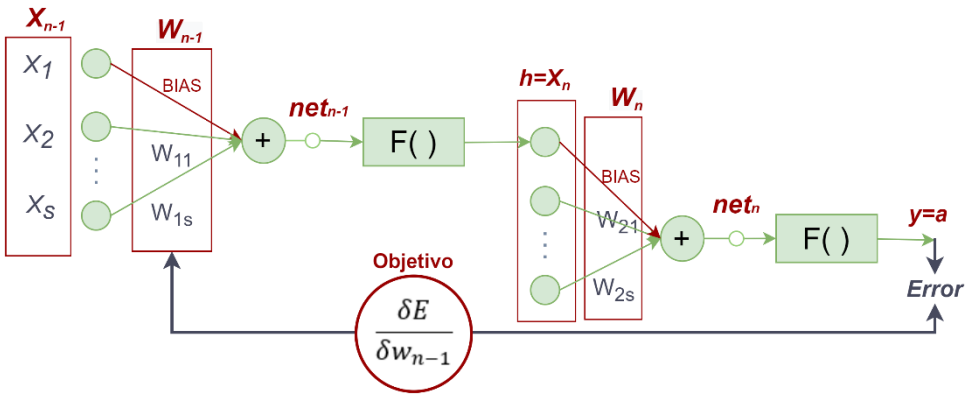


Figura 1.8 Se busca minimizar la función de error respecto de los pesos de la capa oculta.[9]

Para cumplir con este objetivo, el problema se lo simplifica en pequeños pasos como se observa en la Figura 1.9:

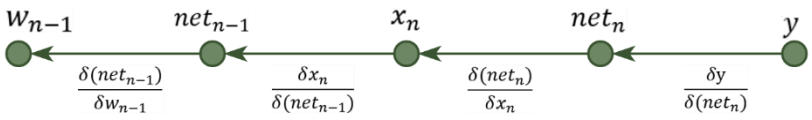


Figura 1.9 Pasos para minimizar la función de error.

$$\frac{\delta E}{\delta w_{n-1}} = \frac{\delta E}{\delta y} \cdot \frac{\delta y}{\delta(net_n)} \cdot \frac{\delta(net_n)}{\delta x_n} \cdot \frac{\delta x_n}{\delta(net_{n-1})} \cdot \frac{\delta(net_{n-1})}{\delta w_{n-1}} \quad (1.18)$$

CÁLCULO DE CADA UNO DE LOS TÉRMINOS

Por la definición de error cuadrático ya visto en la etapa anterior:

$$\frac{\delta E}{\delta \mathrm{y}} = -2 \cdot (\mathrm{t} - \mathrm{y}) \qquad (1.19)$$

De la Figura 15.17 se puede apreciar directamente que:

$$\frac{\delta y}{\delta(net_k)} = f'(net_k) \qquad (1.20)$$

Ya que $net_k = \sum x_k \cdot w_k$:

$$\frac{\delta(net_k)}{\delta(x_k)} = w_k \qquad (1.21)$$

También se puede apreciar directamente que:

$$\frac{\delta x_k}{\delta(net_{k-1})} = f'(net_{k-1}) \qquad (1.22)$$

Finalmente, ya que $net_{k-1} = \sum x_{k-1} \cdot w_{k-1}$:

$$\frac{\delta(net_{k-1})}{\delta w_{k-1}} = x_{k-1} \qquad (1.23)$$

Uniendo todos estos resultados se tiene que la **regla para el aprendizaje para capa oculta** es:

$$\frac{\delta E}{\delta w_{k-1}} = -2 \cdot (\mathrm{t} - a) \cdot f'(net_k) \cdot w_k \cdot f'(net_{k-1}) \cdot x_{k-1} \qquad (1.24)$$

Como la actualización de los pesos tiene la variación contraria al error:

$$w(k+1) - w(k) = -\frac{\delta E}{\delta net_{k-1}} \qquad (1.25)$$

Entonces, la actualización de pesos se daría de la siguiente manera:

$$w(k+1) = w(k) + 2 * (\mathrm{t} - a) * f'(net_k) * w_k * f'(net_{k-1}) * x_{k-1} \qquad (1.26)$$

Agregando el learning rate:

$$w(k+1) = w(k) + 2 \cdot Lr.(t - y) \cdot y(1 - y) \cdot w_k \cdot h(1 - h) \cdot x_{k-1} \qquad (1.27)$$

$w_k = w_2$ serían los pesos ya entrenados en el paso anterior

$x_{k-1} = x_1$ son las entradas conocidas

Cabe resaltar que el entrenamiento de una neurona no afecta al entrenamiento que se haya realizado anteriormente con otras, es decir que se puede considerar a cada neurona como un problema independiente.

1.3.2.1.5. Escalado de valores de entrada

Los datos de entrenamiento se escalan por dos razones:

- Primero, los datos de entrada generalmente se escalan para darle a cada entrada la misma importancia y para evitar la saturación prematura de las funciones de activación sigmoidea.
- En segundo lugar, los datos de salida u objetivo se escalan si las funciones de activación de salida tienen un rango limitado y los objetivos sin escala no coinciden con ese rango.

Hay dos tipos populares de escala de entrada: escala lineal y z-score. El escalado lineal transforma los datos en un nuevo rango que suele ser de 0,1 a 0,9. Si los patrones de entrenamiento tienen una forma tal que las columnas son entradas y las filas son patrones.

El **escalado lineal** transforma los datos en un nuevo rango que suele ser de 0,1 a 0,9. Conservando la distribución original.

Estandarización z-score o escala de varianza de la unidad central media. Este método resta la media de cada entrada de cada columna y luego divide por la varianza. Esto centra todos los patrones de cada tipo de datos alrededor de 0 y les da una varianza unitaria. En la Figura 1.10 se puede observar la distribución de los datos cuando se hace una estandarización z-score.

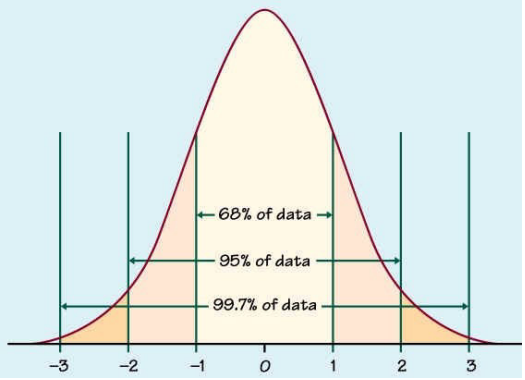


Figura 1.10 Representación de z-score [11]

1.3.2.2. Support Vector Machine (SVM)

SVM (Máquinas de soporte Vectorial), es un algoritmo supervisado utilizado para aplicaciones regresión y clasificación, normalmente es más usado para esta última. Este es muy usado debido a que es muy robusto y se basa en marcos de aprendizaje estadístico. Su modo de funcionamiento consiste en que se ingresa un conjunto de entrenamiento, se crea un modelo SVM basado en los datos, clasificando los datos con su

respectiva clase, finalmente se pueden hacer predicciones para nuevos datos. Cabe recalcar que este clasificador es binario, por lo tanto, en futuras secciones se verá métodos para usarlo con varias clases.

En la Figura 1.11, se crea un hiperplano para clasificar los puntos rojos de los puntos verdes, como se puede ver se puede trazar n líneas, pero sólo una de ellas es ideal para particionar correctamente las dos clases. Se observa en la Figura 1.11 donde la línea celeste se encuentra más cerca de la clase roja por lo tanto no se tiene una generalización adecuada. SVM trata de encontrar el hiperplano que permita la mejor generalización.

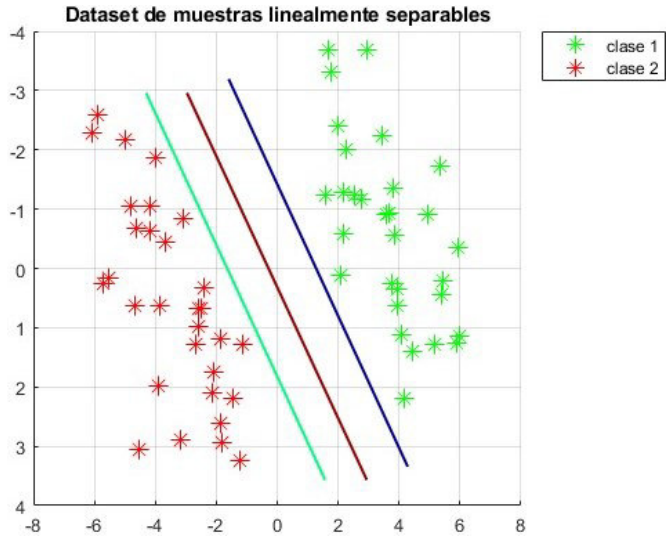


Figura 1.11 Datos linealmente separables separados por varias fronteras de decisión

1.3.2.2.1. SVM lineal

Para explicar el algoritmo SVM, primero se debe empezar con un caso simple, para ello se supone que existen dos clases que son linealmente separables, como en la Figura 1.12.

Sea el conjunto $(x_1, y_1), \dots, (x_D, y_D)$, donde x_i es el conjunto de muestras asociadas con las etiquetas de clase y_i, donde cada y_i puede tomar +1 o -1, D es el número de características o número de dimensiones. Se dice que hay un infinito número de líneas o hiperplanos que separan en 2 clases, pero se quiere encontrar el hiperplano que sea tenga menos errores a la hora de clasificar.[12]

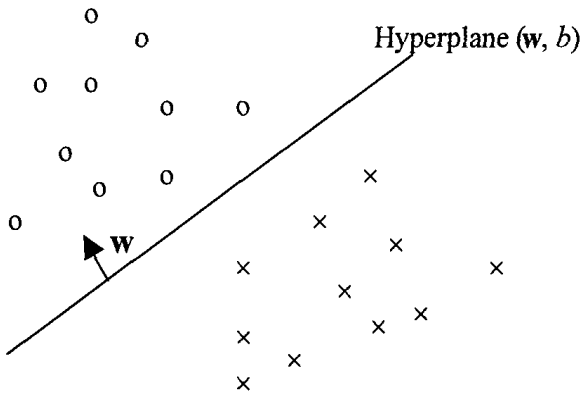


Figura 1.12 Datos linealmente separables separados por una frontera de decisión. [13]

De acuerdo con la Figura 1.12, el término $\boldsymbol{W}$ representa un vector perpendicular al hiperplano, y el término b puede variar la posición del plano en paralelo a sí mismo.

El hiperplano puede ser descrito como:

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0 \tag{1.28}$$

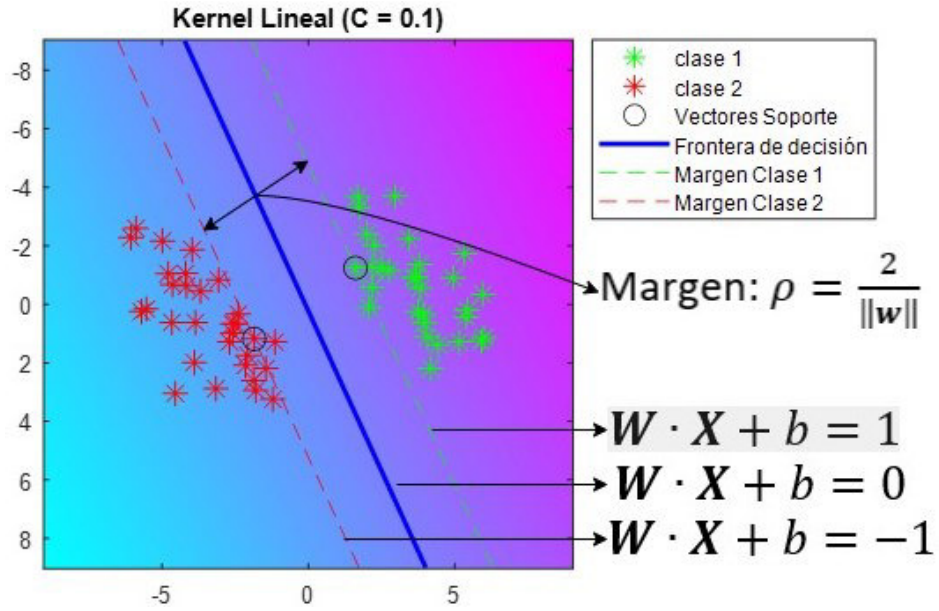


Figura 1.13 Elementos del algoritmo SVM

De la Figura 1.13, se observa que para calcular el margen se debe aplicar la ecuación de distancia de un punto a un plano, si se desarrolla la expresión se llega a la ecuación **(1.21**).

La expresión matemática del margen se representa como:

$$\rho = \frac{2}{\|\boldsymbol{w}\|} \tag{1.29}$$

Margen

La distancia entre los puntos y la línea divisoria se conoce como margen. El objetivo de un algoritmo SVM es maximizar este mismo margen, cuando alcanza su máximo, la línea que divide las clases se convierte en la óptima. [12]

Vectores de Soporte

Los vectores de soporte, hablando geométricamente, en general son los vectores más cercanos a la frontera de decisión al hiperplano, estos juegan un papel importante en la construcción de los algoritmos de aprendizaje en el algoritmo SVM, se puede apreciarlos en la Figura 1.13.

Para tener la mejor frontera de decisión se debe maximizar el margen lo mejor posible, por lo tanto, el problema de optimización se puede desarrollar de la siguiente manera:

1) De la ecuación **(1.30**) se tiene que maximizar el margen por lo que se llega a la siguiente expresión:

$$\max_{x} \frac{2}{\|\boldsymbol{w}\|} \qquad (1.30)$$

2) Para maximizar la expresión anterior se minimiza $\|\boldsymbol{w}\|$, por lo tanto, se reescribe el problema de la siguiente manera:

$$\min_{w} \|\boldsymbol{w}\| \qquad (1.31)$$

3) Si se eleva la norma al cuadrado y se divide para 2, se tiene:

$$\min_{w} \mathbf{1}/_{\mathbf{2}}\, \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} \qquad (1.32)$$

Las expresiones anteriores se reducen un problema de optimización y se pueden reescribir de la siguiente manera:

$$minimizar_{\boldsymbol{w}}\, \boldsymbol{\Phi}(\boldsymbol{w}) = \mathbf{1}/_{\mathbf{2}}\, \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} \qquad (1.33)$$

Hard Margin

Si se tiene un caso donde existen datos linealmente separables como en la Figura 1.14, en este caso ideal y sin ruido se tienen a los vectores justo en los límites del margen, este caso se llama Hard margin debido a que los vectores de soporte no deben estar dentro del margen. El problema para este caso sucede cuando existe la presencia de ruido en las muestras el algoritmo va a tender un margen muy pequeño y tiende al sobreajuste.[12]

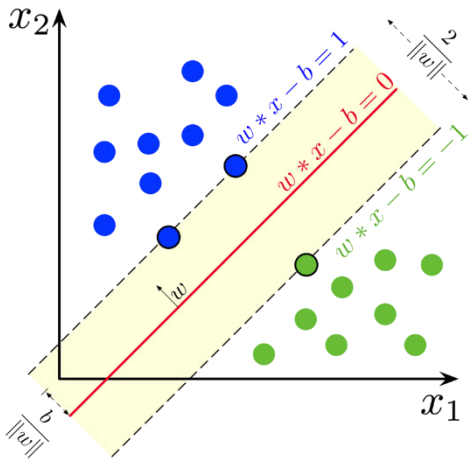


Figura 1.14 Datos linealmente separables, separados por una frontera de decisión con sus márgenes. [14]

Los hiperplanos están definidos por:

$$H_1: \boldsymbol{w} \cdot \boldsymbol{x} + b = 1 \quad (1.34)$$

$$H_2: \boldsymbol{w} \cdot \boldsymbol{x} + b = -1 \quad (1.35)$$

Además de maximizar el margen entre los hiperplanos que definen cada clase, se deben seguir las siguientes condiciones:

$$\boldsymbol{w} \cdot \boldsymbol{x} - b \geq 1 \text{ si } y_i = 1 \quad (1.36)$$

$$\boldsymbol{w} \cdot \boldsymbol{x} - b \geq -1 \text{ si } y_i = -1 \quad (1.37)$$

Al combinarse las dos inecuaciones ecuación **(1.36**) y ecuación **(1.37**) se obtiene:

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x} - b) \geq 1, \qquad \forall\, i \quad (1.38)$$

Finalmente, se llega a un problema de optimización:

$$\begin{array}{ll} minimizar_{\boldsymbol{w},b} & \boldsymbol{\Phi}(\boldsymbol{w}) = {}^{\boldsymbol{1}}/_{\boldsymbol{2}}\, \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} \\ sujeta\ a & y_i(\boldsymbol{w} \cdot \boldsymbol{x} + b) \geq 1 \end{array} \quad (1.39)$$

La solución a este problema se verá más adelante.

Soft Margin

Este método se basa en una simple idea: permitir al algoritmo un número determinado de errores para mantener el margen lo más amplio posible para que otros puntos puedan ser clasificados correctamente, a un cierto costo. Las variables holgura están representadas por ξ_i, se puede apreciar en la Figura 1.15. El método es el más usado debido a que en la vida real no se tienen datos completamente linealmente separables.[12]

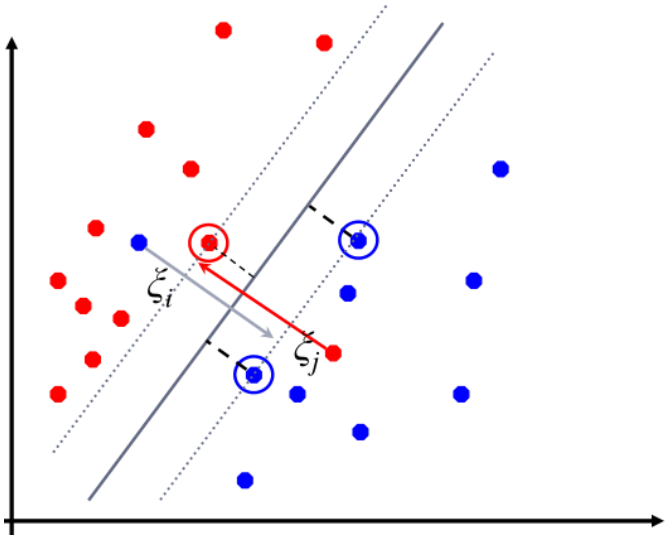


Figura 1.15 Datos linealmente separables, separados por un hiperplano en donde se encuentran variables linealmente separables. [15]

Bajo la idea del anterior párrafo, y dada la Ecuación (**1.39**), se tiene una nueva fórmula con variables de holgura:

$$\begin{aligned} &minimizar_{\boldsymbol{w}} && \boldsymbol{\Phi}(\boldsymbol{w}) = {}^{\mathbf{1}}\!/_{\mathbf{2}}\, \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} + \boldsymbol{C}\sum \xi_i \\ &sujeta\ a && y_i(\boldsymbol{w}\cdot\boldsymbol{x}+b) \geq 1-\xi_i,\ \ \xi_i \geq 0\,, \qquad \forall\, i \end{aligned} \tag{1.40}$$

1.3.2.2.2. Cálculo de SVM

En la Ecuación 1.40, el problema que genera es cuadrático con restricciones de desigualdad lineal, por lo que es un problema de optimización convexa. Se describe una solución de programación cuadrática utilizando multiplicadores de Lagrange. Computacionalmente es conveniente volver a expresarla en la forma equivalente.

El problema puede ser presentado de dos formas de manera primal y de manera dual, la primera está presente en las Ecuación (**1.39**) y Ecuación (**1.40**). Aunque la forma más común esta expresada con ayuda de multiplicadores de Lagrange, que se verá a continuación.

Multiplicadores de Lagrange

Antes se debe hacer un breve repaso de los multiplicadores de Lagrange.

Entonces, para un problema de optimización se tiene:

$$\min_x F(x)$$

Sujeto a:

$$g_i(x) = 0\ para\ i\ = 1,2,\dots n$$

Se puede convertir el problema a:

$$\max_{x,\alpha} f(x) - \sum_{i=i}^{n} \alpha_i g_i(x) \qquad (1.41)$$

Forma Dual

Entonces basándose en la Ecuación (**1.41**), el problema dual para la clasificación Soft Margin:

Se debe ver que $\boldsymbol{w}, b, \alpha$ cumplir con las condiciones KKT, y después de hacer unas simplificaciones se tiene que encontrar $\alpha_1 \dots \alpha_n$ tal que:

$$\begin{aligned} maximizar \quad & Q(\alpha) = \sum \alpha_i - \frac{1}{2} \sum \sum \alpha_i \alpha_j y_i y_j \boldsymbol{x}^{\boldsymbol{T}} \times \boldsymbol{x} \\ sujeta\ a \quad & \sum \alpha_i y_i = \mathbf{0} \\ & 0 < \alpha_i \leq C, \qquad i = 1, \dots, N \end{aligned} \qquad (1.42)$$

α_i, α_j: son multiplicadores de Lagrange que se debe encontrar.

Como se puede observar en la Ecuación (**1.42**), se ve que la variable ξ_i, que representa a la holgura, no aparece ni entre sus multiplicadores de Lagrange, lo que nos sugiere que no depende de esta variable la resolución del problema.

1.3.2.2.3. La solución del problema de optimización

Los problemas de optimización cuadrática son una clase muy conocida de problemas de programación matemática, y existen muchos algoritmos para resolverlos. La solución implica la construcción de un multiplicador de Lagrange α_i, que está asociado con cada restricción del problema principal.

1.3.2.2.4. SVM no lineal

Como se puede observar en la Figura 1.16, el cerebro humano puede reconocer a los puntos que están en un círculo. El clasificador por otro lado solo puede reconocer si se toma un valor inferior o superior a cero de una frontera de decisión lineal, por lo tanto, los datos de un espacio funciones se deben transformar a otro para que estos puedan ser manejados fácilmente con las herramientas matemáticas que se tiene. A este proceso se le llama mapeo y este se encarga de ir de un espacio de características de menor dimensión a un espacio de mayor dimensión. Este mapeo se realiza con Kernel, entonces se puede pensar en un Kernel como un contenedor de interfaz para que los datos sean traducidos de un formato más difícil a otro más fácil.

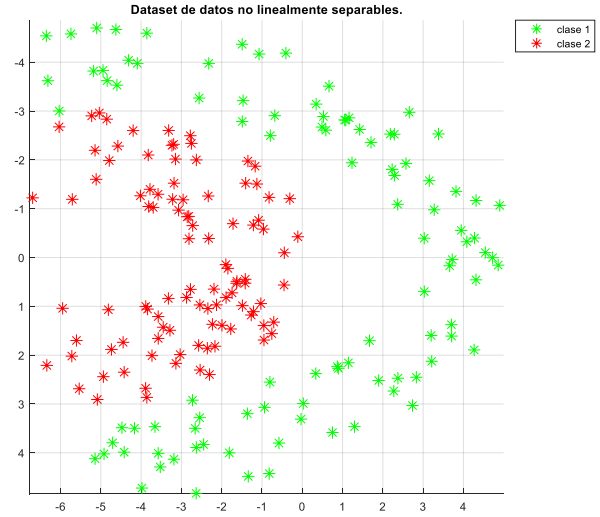


Figura 1.16 Datos no linealmente separables.

Como se puede apreciar en la Figura 1.17 si bien se tiene variables de holgura estas no son suficientes para clasificar correctamente las muestras. En la siguiente sección se verá cómo solucionar este problema. Al correr el programa completo anterior se obtiene lo siguiente:

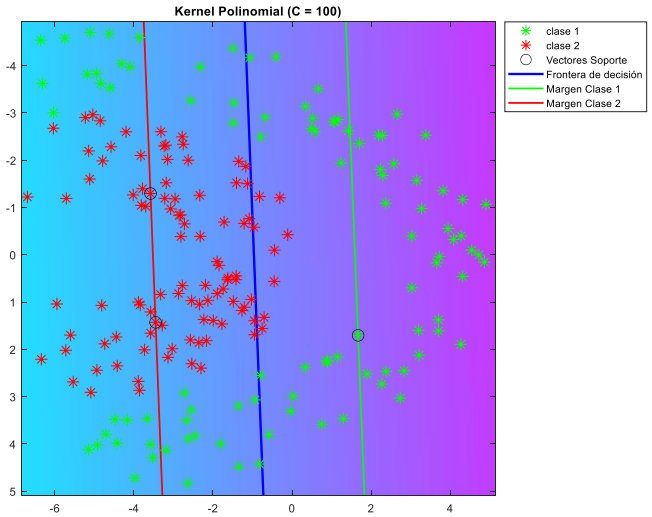


Figura 1.17 Datos no separables linealmente, tratando de ser separados por el algoritmo SVM con kernel lineal.

Uno de los aspectos positivos de la optimización de SVM es que las operaciones se pueden escribir en términos de productos internos estos se multiplican para producir un escalar único, reemplazar el producto interno con un kernel este sí conoce como el truco o método de kernel. En la Figura 1.18 se puede observar de manera gráfica cómo funciona el kernel.

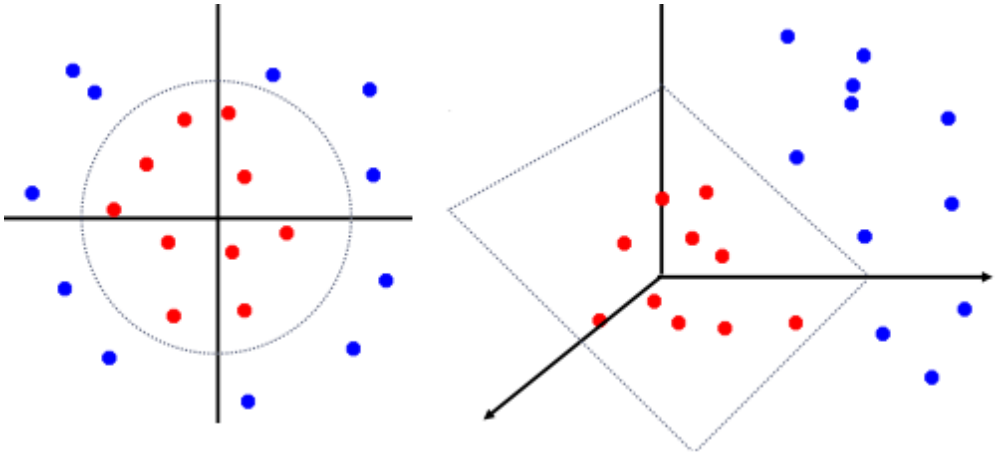


Figura 1.18 Ejemplo del funcionamiento de la función de kernel.[15]

Método de kernel

El clasificador lineal se basa en un producto interno entre los vectores $K(\boldsymbol{X^T}, \boldsymbol{X}) = \boldsymbol{X^T} \times \boldsymbol{X}$

Si cada punto de datos se mapea en un espacio de alta dimensión mediante alguna transformación$\Phi: \mathrm{x} \rightarrow \varphi(\mathrm{x})$, el producto interno se convierte en:

$$K(\boldsymbol{X^T}, \boldsymbol{X}) = \Phi(\boldsymbol{X})^T \Phi(X) \qquad (1.43)$$

Entre los distintos kernels que se puede encontrar están:

Kernel lineal

Son más utilizados en problemas lineales, son los más básicos con los que se puede trabajar. Matemáticamente se define como:

$$K(\boldsymbol{X^T}, \boldsymbol{X}) = \boldsymbol{X^T} \times \boldsymbol{X} \qquad (1.44)$$

Su código en Matlab está dado por:

```
K = X'*X;
```

Y sus resultados están presentes en la Figura 1.18 Es decir, el kernel lineal es la base de partida para los demás.

Kernel polinomial

Estos están definidos sobre el espacio vectorial X de dimensión, donde r normalmente es 1 y d es el grado del polinomio. En la Figura 1.19 se puede ver una representación del kernel polinomial. Matemáticamente se define como:

$$K(\boldsymbol{X^T}, \boldsymbol{X}) = (\boldsymbol{X^T} \times \boldsymbol{X} + r)^d \qquad (1.45)$$

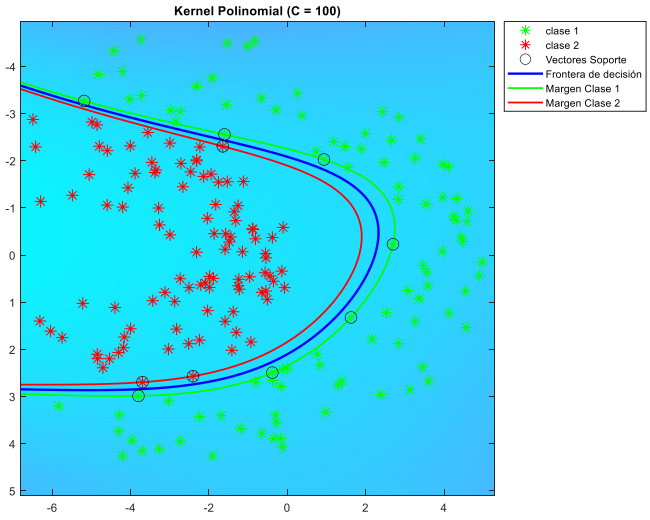


Figura 1.19 SVM con kernel polinomial.

Kernel de base radial gaussiana

Estos son los más estudiados y ampliamente utilizados.El parámetro γ controla la flexibilidad del kernel de igual forma que el grado d de los polinomiales. En la Figura 1.20 se puede ver una representación del kernel polinomial . Matemáticamente se define como:

$$K(\boldsymbol{X^T}, \boldsymbol{X}) = e^{-\gamma \|\boldsymbol{X^T} - \boldsymbol{X}\|^2} \qquad (1.46)$$

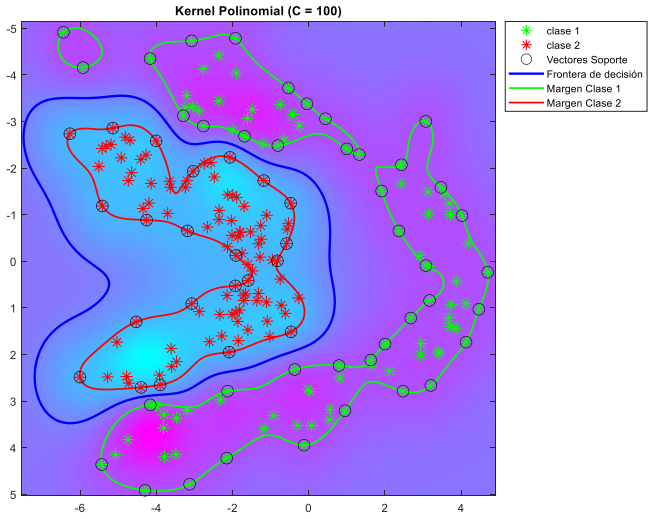


Figura 1.20 SVM con kernel de base radial gaussiana.

1.3.2.2.5. SVM Multiclase

El problema con el algoritmo SVM es que no admite clasificación multiclase, es decir que se limita a una clasificación binaria. Pero la solución se encuentra en dividir el problema en varios problemas de clasificación binaria. La solución se puede dar de dos modos, "*One vs One*" y "*One vs All*", estos se verán a continuación. [16] En la Figura 1.21 se observa un ejemplo sencillo de un conjunto de datos de 3 clases.

Figura 1.21 Conjuntos de datos separados en varias clases. [17]

One vs One (OvO)

Para este enfoque se necesita un hiperplano para cada dos caso como se observa en la Figura 1.22, descuidando los puntos de la tercera clase, Esto significa que la separación toma en cuenta sólo los puntos de la clasificación actual. [18]

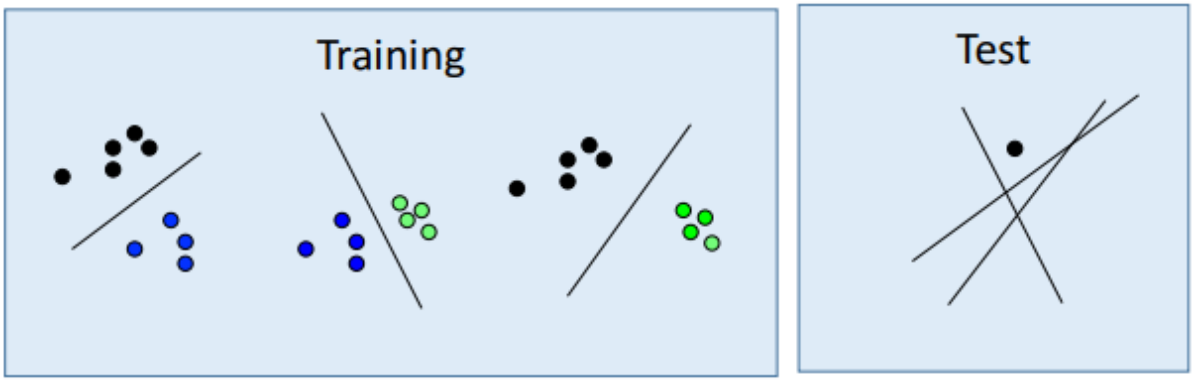


Figura 1.22 Método SVM One vs One [17]

One vs All (OvA)

Este enfoque consiste en separar una clase de todas las demás, esto significa que la separación toma en cuenta todas las muestras, dividiéndolos en dos grupos, uno para los de la clase elegida y otro para las demás muestras, como se indica en la Figura 1.23.

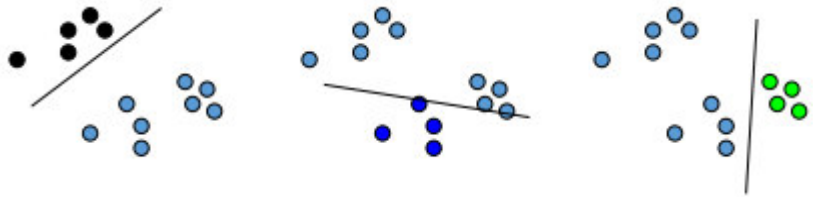


Figura 1.23 Método SVM One vs All. [17]

Se descompone en algunos clasificadores binarios[19]

¿Cómo funciona el algoritmo?

a) Dado un conjunto de datos: $D = \{(x_i, y_i)\}$

b) Donde $x_i \in R^n, y_i \in \{1, 2, \dots, K\}$

c) A continuación, se descompone en K clasificadores binarios

d) Se entrenan K modelos: $w_1, w_2, \dots, w_K$

e) Para cada clasificador binario se tiene:

- Muestras positivas: elementos de D con etiqueta k
- Muestras negativas: todos los demás elementos

f) El Clasificador binario se resuelve por cualquier método.

Para decidir en qué clase va, se usa la siguiente expresión:

$$F(x) = \arg max f_t(x) \quad (1.47)$$

Quiere decir que, al evaluar la muestra de prueba en todos los clasificadores, por ejemplo, clasificador 1, clasificador 2 y clasificador 3, se comparan los tres si en el clasificador 1 se tiene el mayor valor de salida entonces la muestra pertenece a la clase 1, si en los resultados del clasificador 3 se tiene el valor más alto de la salida entonces pertenece a la clase 3.

1.3.2.3. Decision Tree (DT)

La técnica de machine learning denominada “Decisión Tree”, es un algoritmo relativamente simple que de preferencia es utilizado para resolver problemas de clasificación. Su objetivo es crear un modelo de entrenamiento que se pueda usar para predecir el valor de una variable nueva, entrenado previamente con un conjunto de datos.

Para lograr su cometido el árbol de decisión se parte de la raíz, con base a las características de los datos el modelo aprende una serie de preguntas para inferir las etiquetas de clase de muestras. [20]

¿Por qué elegir un árbol de decisión?

Haya varios algoritmos de clasificación de Machine Learning, por lo que elegir el mejor algoritmo depende del conjunto de datos y del problema para crear un modelo, a continuación, se tiene dos buenas razones para usar el árbol de decisión:

- Los árboles de decisión imitan el pensamiento del ser humano al tomar una decisión, por lo que se vuelve fácil de entender. [21]
- La lógica detrás del árbol se puede entender fácilmente debido a que su estructura se asemeja a la de un árbol. [21]

1.3.2.3.1. Tipos de atributos que usan árboles de decisión

Los árboles de Decisión se pueden clasificar de acuerdo con el tipo de atributos que manejan, estos pueden ser:

Variables categóricas: son variables que pueden pertenecer a distintos grupos finitos, que tienden a ser excluyentes, estos pueden tener o no un orden lógico.

Ejemplos de variables categóricas: tipo de material, género de una persona, método de pago.

Variables discretas: son variables numéricas, que pueden tomar un valor fijo dentro de un rango determinado.

Ejemplos: número de quejas, número de hijos, número de fallas.

Variables continuas: se refieren a árboles que tienen variables continuas, es decir que las variables pueden tomar cualquier valor en un intervalo. Son usados en casos de regresión.

Ejemplos de variables cuantitativas: temperaturas registradas en un observatorio.

1.3.2.3.2. Terminología relacionada al árbol de decisión

Nodo raíz: representa a toda la muestra que se divide en varios conjuntos.

División: se refiere al proceso de división de un nodo en 2 o más subnodos.

Nodo de decisión: un subnodo que se divide en varios nodos se dice nodo de decisión.

Nodo hoja o terminal: se refiere a los nodos que ya no se dividen más se denominan.

Poda: se refiere cuando se eliminan un nodo de decisión, es el proceso contrario a la división. Se puede ver con más claridad de que se trata está en la sección para contrarrestar el sobreajuste en los árboles de decisión.

Rama o sub-árbol: es una subsección de todo el árbol.

Nodo principal/ nodo secundario: el nodo raíz se denomina nodo principal mientras que los demás se denominan nodos secundarios.

En la Figura 1.24 se puede observar un árbol de decisión asociado a cada terminología que vimos anteriormente.

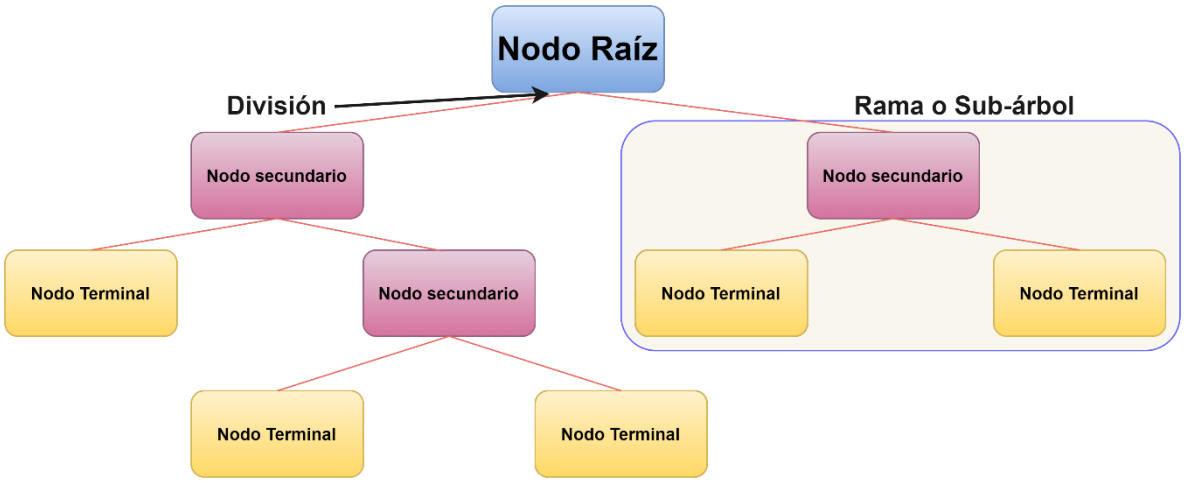


Figura 1.24 Elementos de un árbol de decisión. [22]

1.3.2.3.3. Medidas de selección de atributos para decidir entre 2 caminos

Si se tiene n atributos, entonces se debe decidir qué atributo se debe colocar o en diferentes niveles del árbol, como se mencionó anteriormente este es un paso complicado, debido a que si se trata de elegir aleatoriamente cualquier nodo para que sea la raíz se tendrá malos resultados con poca precisión. Para resolver este problema se idearon los siguientes criterios:

- Entropía
- Ganancia de información

Estos criterios calculan valores para cada atributo, los valores se ordenan y se colocan en el árbol según el orden eso significa que un atributo alto estará en la raíz.

Entropía

La entropía es una medida de aleatoriedad de información, mientras mayor sea esta más difícil es sacar conclusiones de esa información. Matemáticamente la entropía de un atributo esta dado por:

$$E(S) = -\sum_{i=1}^{C} p_i \log_2 p_i \qquad (1.48)$$

Donde:

S: es el estado actual

P_i: probabilidad de un evento i en el estado S

Ejemplo, lanzar una moneda al aire nos proporciona una información aleatoria.

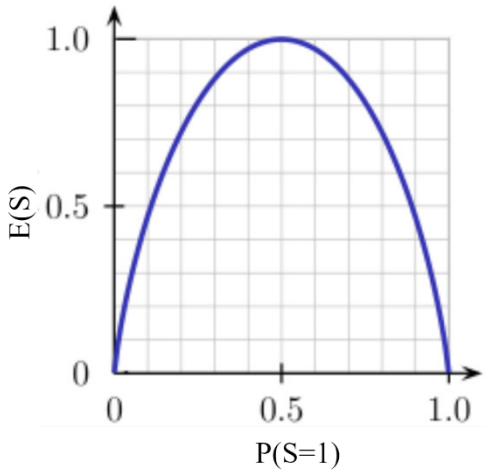


Figura 1.25 Entropía vs probabilidad. [22]

Como se puede observar en la Figura 1.25 la entropía en cero cuando la probabilidad es cero o uno, pero es máxima cuando esta es 0.5, entonces en los árboles de decisión el atributo con mayor entropía para será considerado más cercano al nodo raíz para clasificar los datos. [20]

Ganancia de información

La ganancia de información es básicamente una medida de cambio de la entropía, esta mide que tan bien un atributo determinado separa los ejemplos de entrenamiento de acuerdo con su clasificación de destino. La construcción de un árbol de decisión se basa en que se devuelva la mayor ganancia de información y la menor entropía. Un ejemplo de ganancia de información se puede apreciar en la Figura 1.26.

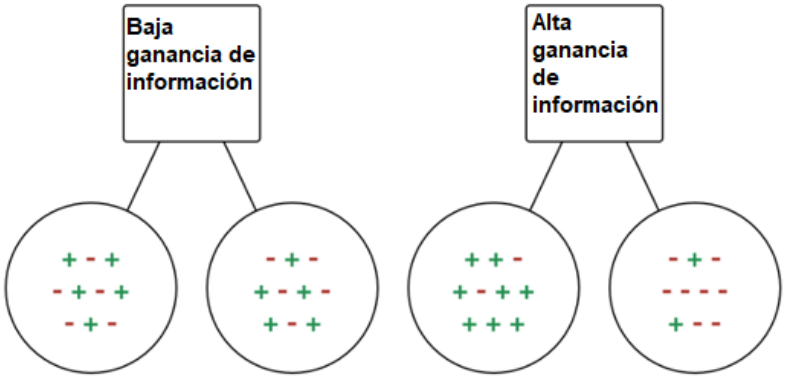


Figura 1.26 Ganancia de Información. [22]

La ganancia de la información es una disminución de la entropía, esta se calcula de la diferencia entre la entropía antes de la división y la entropía promedio después de la división del conjunto de datos. Esta se representa matemáticamente como:

$$Ganacia\ Información = Entropia(antes) - \sum_{j=1}^{K} Entropia(j, después) \quad (1.49)$$

Donde se puede ver que “antes” es el conjunto antes de la división, K es el número de subconjuntos generados por la división, y (j, después) es el j conjunto después de la división.

Chi-cuadrado

Es un parámetro que descubre la significancia estadística entre diferencias de subnodos y el nodo principal. Lo medimos por la suma de cuadrados de las diferencias entre diferencias estandarizadas entre las frecuencias observadas y esperadas de la variable objetivo. funciona normalmente con la variable categórica “éxito” o “falla”, puede realizar dos o más divisiones. Cuanto mayor sea su valor mayor significación estadística de las diferencias entre el subnodo principal y el sub-nodo secundario.

$$x^2 = \sum \frac{(O - E)^2}{E} \quad (1.50)$$

Donde:

x^2: Chi-cuadrado obtenido

O: valor observado

E: valor esperado

Pasos para calcular chi-cuadrado de una división

Se calcula chi-cuadrado para un nodo individual, calculando la desviación para el éxito y el fracaso.

Las medidas para elegir atributos como el índice de Gini, la ganancia de información, la relación de ganancia, el Chi-cuadrado, que se usarán varios algoritmos, pero C4.5 solo utiliza el criterio de entropía y de ganancia de información para crear el árbol.

1.3.2.3.4. Algoritmo C4.5

Los árboles de decisión generados por C4.5 se pueden utilizar para la clasificación, por esta razón, a menudo se denomina clasificador estadístico. Este utiliza la **ganancia de información como criterio de división.** Puede aceptar datos de tipo tanto categóricos, discretos o continuos. Para manejar valores cuantitativos, genera un umbral, luego clasifica las muestras según el umbral. C4.5 puede manejar fácilmente valores perdidos.[23]

1.3.2.3.5. ¿Cómo funcionan los árboles de decisión?

En un árbol de decisión, para predecir la clase del conjunto de datos dado, el algoritmo comienza desde el nodo raíz del árbol. Este algoritmo compara los valores del atributo raíz con el atributo de del conjunto de datos real y, según la comparación, sigue la rama y salta al siguiente nodo.

Para el siguiente nodo, el algoritmo vuelve a comparar el valor del atributo con los otros subnodos y avanza. Continúa el proceso hasta que llega al nodo de la hoja del árbol. El proceso completo se puede comprender mejor utilizando el siguiente algoritmo:

Paso 1: Se comienza el árbol con el nodo raíz, que contiene el conjunto de datos completo.

Paso 2: Se busca el mejor atributo en el conjunto de datos usando la medida de selección de atributos.

Paso 3: Se genera el nodo del árbol de decisión, que contiene el mejor atributo.

Paso 4: Se crea de forma recursiva nuevos árboles de decisión utilizando los subconjuntos del conjunto de datos creado en el Paso 3, posteriormente se repite este proceso hasta que se alcance una etapa en la que no pueda clasificar los nodos más y se llame al nodo final como un nodo hoja.

Supuestos al crear un árbol de decisión

Al crear los árboles de decisión se debe tomar en cuenta las siguientes suposiciones:

- Al principio, todo el conjunto de entrenamiento se le considera como la “raíz”.
- Se refiere que los valores de las características sean discretas, si no los son entonces se las discretiza.

Los registros se distribuyen de manera recursiva sobre la base de los valores de los atributos.

El mayor desafío es identificar el atributo para el nodo en cada nivel.

Poda de árboles de decisión

La poda es un proceso que elimina los nodos innecesarios de un árbol para obtener un árbol de decisión óptimo.

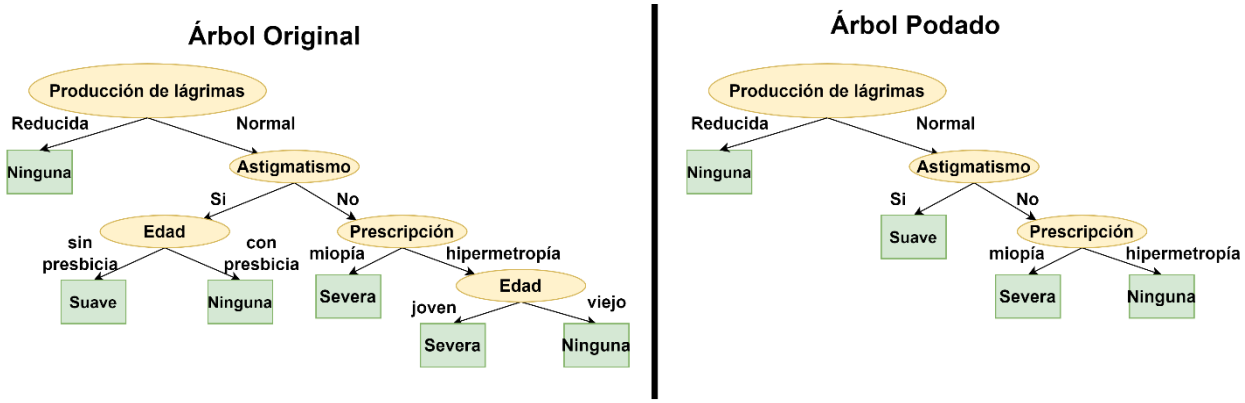


Figura 1.27 Árbol Original y árbol podado. [24]

En la Figura 1.27 se puede ver un ejemplo de la poda de un árbol que tiene un nodo innecesario, una vez podado se elimina el sobreajuste.

Para este caso cuando el nodo tenga una cantidad de datos menor al porcentaje de la cantidad de datos totales, no se dividirá más debido, para evitar árboles con demasiadas ramas, además de que evita el sobreajuste. Un ejemplo: este dataset muestras conocidas tiene 1000 muestras, con un error del 5%. Cuando se tenga un nodo en el que participan 100 datos el nodo se podrá dividir sin complicaciones debido a que 100 representa el 10% de los datos, pero si se tiene 40 datos en un nodo no se podrá dividir debido a que el 5% equivalen a 50, entonces si el nodo tiene menor a esta cantidad será indivisible.

1.3.2.4. Naïve Bayes (NB)

Naïve Bayes es un algoritmo probabilístico de aprendizaje automático basado en el Teorema de Bayes, utilizado en una amplia variedad de tareas de clasificación.

Las soluciones más simples suelen ser las más poderosas, y Naïve Bayes es un buen ejemplo de ello. A pesar de los avances en machine learning en los últimos años, ha demostrado no solamente ser simple, sino también rápido, preciso y confiable. Se ha utilizado con éxito para muchos propósitos, pero funciona particularmente bien con problemas de procesamiento de lenguaje natural (PNL).[25]

1.3.2.4.1. Teorema de Bayes

En probabilidad el teorema es planteada por Thomas Bayes este nos dice:

Sea $\{A_1, A_2, \dots A_i, \dots A_n\}$ un conjunto de sucesos mutuamente excluyentes tales que la probabilidad de cada uno de ellos es diferente de cero, Si B es un suceso cualquiera que se conocen las probabilidades condicionales $P(B|A_i)$ entonces la probabilidad $P(A_i|B)$ viene dada por la expresión:

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)} \quad (1.51)$$

Donde:

$P(A_i)$: es la probabilidad de que ocurra A_i

$P(B|A_i)$: es la probabilidad de que ocurra B dado que haya ocurrido A_i.

$P(A_i|B)$: es la probabilidad de que ocurra A_i dado que haya ocurrido B.

1.3.2.4.2. Naïve Bayes Gaussiano

En Bayes Naive gaussiano, es una variante que se supone que los valores continuos asociados con cada entidad se distribuyen de acuerdo con una distribución Gaussiana. Cuando se gráfica, da una curva en forma de campana que es simétrica sobre la media de los valores de la entidad como se muestra en la Figura 1.28.

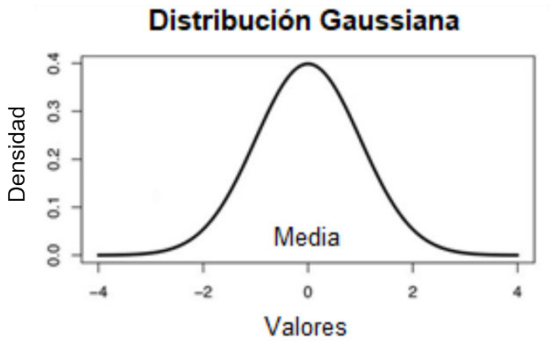


Figura 1.28 Distribución gaussiana. [24]

Se supone que la probabilidad de cada atributo es gaussiana por lo tanto su expresión matemática se presenta como:

$$(X) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (1.52)$$

Donde:

$P(X)$: Probabilidad de que ocurra un evento

μ: Media y es calculada por:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} X_i \quad (1.53)$$

σ: Desviación estándar y es calculada por:

$$\sigma = \left[\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2\right]^{0.5} \quad (1.54)$$

Para afrontar valores numéricos al momento de clasificar se puede transformar estos en muestras categóricas y crear una tabla de frecuencias, pero una opción más factible es usar una buena distribución una distribución normal o Gaussiana.

Pasos del Algoritmo Naive Bayes

1. **Se carga Datos**
2. **Se calcula la probabilidad de que ocurra cada clase.**

$$C_i = \frac{n_i}{S} \quad (1.55)$$

Donde:

C_i: es la probabilidad de que ocurra la clase i.

n_i: es la cantidad de muestras de la clase i.

Ts: número total de muestras

3. **Se calcula la media y la desviación estándar de los atributos por clase.**

$$\mu_i = \frac{1}{n_i}\sum_{k=1}^{n} X_k \quad (1.56)$$

Donde:

μ_i: media de Clase i.

n_i: es la cantidad de muestras de la clase i.

X_k: muestra k de la clase i.

$$\sigma_i = \left[\frac{1}{n_i - 1}\sum_{k=1}^{n}(x_k - \mu_i)^2\right]^{0.5} \quad (1.57)$$

Donde:

σ_i: Representa la desviación estándar de la clase i.

Finalmente, para encontrar la clase a la que pertenece se multiplica $Pxc_{k,i}$ por C_i. Como el denominador en el teorema de Bayes es igual para todos los casos, para clasificar las muestras se tiene la siguiente expresión:

$$E_p = argmax(Pxc_{k,i} * C_i) \quad (1.58)$$

Donde:

E_p: es la etiqueta predicha.

1.3.2.5. K-Nearest Neighbor (KNN)

Este algoritmo es una de las técnicas de minería de datos considerada entre las 10 mejores técnicas de minería de datos.[26] Trata de clasificar una muestra desconocida basándose en la clasificación conocida de sus vecinos.

En la Figura 1.29, en imagen de la izquierda se quiere clasificar a la estrella mientras que en la derecha se ve que se escoge los 3 vecinos más cercanos por lo tanto la estrella pertenecería a la clase de puntos rojos.

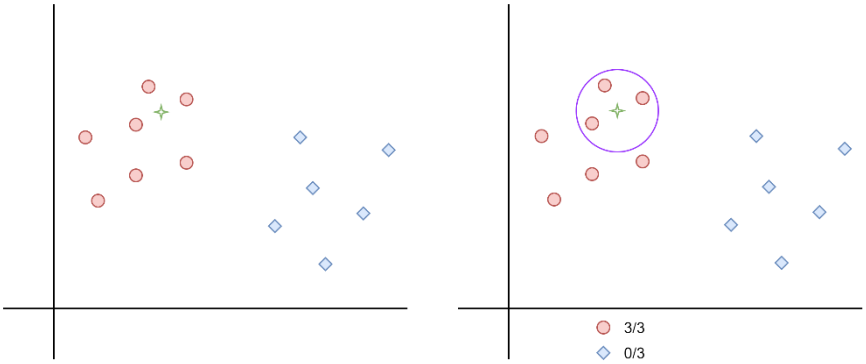


Figura 1.29 Ejemplo básico de clasificación con KNN. [27]

Sin embargo, en general, esta regla de clasificación puede ser débil, porque se basa en una sola muestra conocida. Puede ser precisa si la muestra desconocida está rodeada de varias muestras conocidas con la misma clasificación, pero en la vida real casi no ocurre dicho suceso. Por lo tanto, para aumentar el nivel de precisión, deben considerarse todas las muestras circundantes y la muestra desconocida debe clasificarse en consecuencia. En general, la regla de clasificación basada en esta idea simplemente asigna a cualquier muestra no clasificada la clase que contiene la mayoría de sus k vecinos más cercanos [28]. La Figura 1.30 muestra cómo influye el valor de k en la clasificación de muestras.

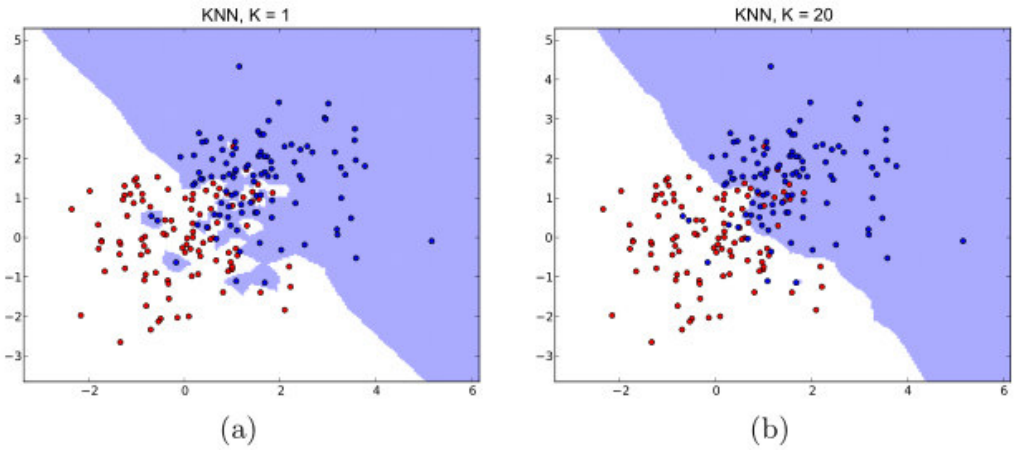


Figura 1.30 Ejemplos cuando K=1 y cuando K=20. [29]

1.3.2.5.1. Aprendizaje con kNN

Como medición de distancia se utiliza la distancia euclidiana.

Este algoritmo asume que todas las instancias corresponden a puntos en un espacio n-dimensional. Los vecinos más cercanos de una muestra se definen en términos de la norma. Primero que una muestra arbitraria x sea descrita por el vector de características:

$x = \langle a_1(x), a_2(x), a_r(x), \dots\ a_n(x) \rangle$

Donde $a_r(x)$ denota el valor del atributo r de la muestra x. La distancia euclidiana entre dos muestras $\bar{x}_q$ y x_i se define como $d(\bar{x}_q, x_i)$, donde:

$$d(x_q, x_i) = \sqrt{\sum_{r=1}^{n} \left(a_r(x_q) - a_r(x_i)\right)^2} \qquad (1.59)$$

Donde:

x_q: representa la muestra que se pretende etiquetar.

x_i: representa a la muestra que tiene su etiqueta ya predicha.

a_r: representa al r-atributo de cada muestra.

En el aprendizaje del vecino más cercano, una vez que se tiene las distancias de las muestras pertenecientes al conjunto de entrenamiento a la muestra que se quiere etiquetar, se busca las k distancias menores, una vez que las se busca se cuenta las etiquetas de estas y la clase que tiene más muestras determina la clase de la muestra que se quiere etiquetar

¿Cómo se escoge k?

No existen métodos estadísticos predefinidos para escoger el valor óptimo de k. Por lo tanto, usualmente se escoge un valor aleatorio y se empieza a calcular, hasta tener una tasa de error mínima. Usualmente si se tiene un valor muy pequeño de k significa que el ruido tendrá mayor influencia en el resultado. Mientras que si se escoge un valor muy grande el algoritmo se hace computacionalmente costoso [30].

Al escoger los datos hay unas reglas que los científicos de datos siguen de manera empírica:

- El valor de k debe ser impar:
- El valor de k no debe ser múltiplo del número de clases.
- El valor de k no debe ser muy grande o pequeño.
- En varios foros también se puede ver que k puede calcularse de la siguiente manera:

$$k = \sqrt{n} \qquad (1.60)$$

Donde:

K: es el número de vecinos más cercanos.

n: es el número total de muestras de entrenamiento.

1.3.2.6. Probabilistic Neural Network (PNN)

Las PNN (Probabilistic Neural Network) son redes neuronales que sirven como una alternativa escalable a las redes neuronales clásicas de retropropagación en aplicaciones de clasificación. Como se puede apreciar en la Figura 1.31, las PNN reemplazan la función de activación sigmoidea que se usa a menudo en las redes neuronales con una función exponencial. Normalmente son usadas utilizando el enfoque de Parzen para diseñar una familia de estimadores de funciones de densidad de probabilidad, usando las denominadas “estrategias de Bayes”. [31]

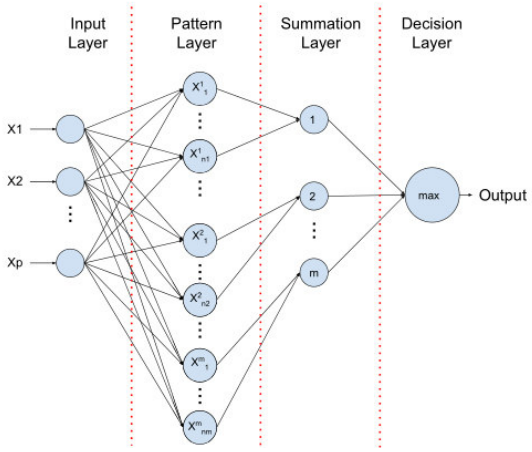


Figura 1.31 Ejemplo de una PNN. [32]

En estas redes no prescindimos del algoritmo de retropropagación. A cambio, se presenta a cada patrón de datos como una unidad que mide la similitud de los patrones de entrada con el de datos. Para comprender la base de este algoritmo, se debe hablar sobre Bayes y los estimadores no paramétricos de las funciones de densidad de probabilidad. Por lo tanto, en los siguientes apartados se verán estos conceptos.

1.3.2.6.1. Estructura de una PNN

En la Figura 1.32 se puede ver que la red tiene cuatro capas entre ellas están:

1) Capa de entrada

2) Capa de patrón

3) Capa de suma

4) Capa de decisión

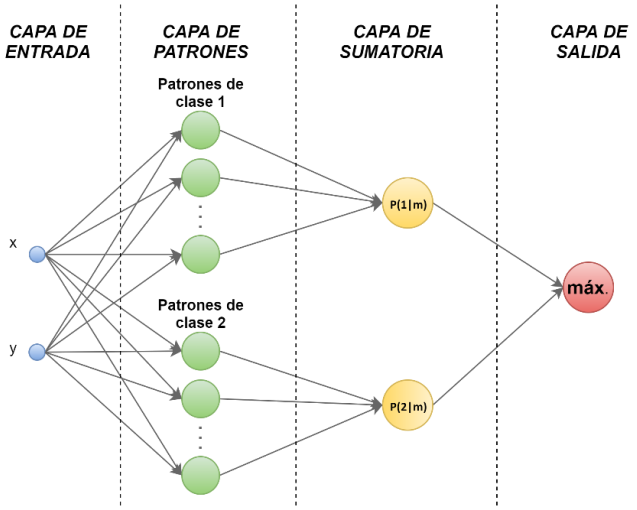


Figura 1.32 PNN usada en este caso. [31]

En la **capa de entrada** cada nodo representa un atributo o dimensión de la capa de entrada.

En la **capa de patrones** se tiene tantas neuronas como muestras de entrenamiento, en cada una se calcula la distancia euclidiana entre la muestra que se quiere predecir y cada muestra de entrenamiento.

En la **capa de Sumatoria** se tiene tantas neuronas como número de clases cada salida de la neurona de la capa de patrones se une con su respectiva clase se calcula usando una función gaussiana y se suma todas las salidas para finalmente tener un valor en su salida que corresponde a la probabilidad de que la muestra pertenezca a la clase correspondiente a la neurona asignada.

En la **capa de salida** se escoge la máxima probabilidad entre las salidas de las dos neuronas de la capa de sumatoria.

En ella en vez de usar una ventana de Parzen se usa una función gaussiana de la distancia euclidiana entre el vector de entrada o el vector del que se pretende conocer su etiqueta y un conjunto de muestras con sus etiquetas ya conocidas. Entonces a la distancia euclidiana la se puede expresar como:

$$D(x_s, x) = \sqrt{\sum_{r=1}^{n} \left(a_r(x_s) - a_r(x)\right)^2} \qquad (1.61)$$

Donde:

$a_r(x_s)$: representa el r-atributo o dimensión de la muestra x_s que se planea predecir su etiqueta, para este caso se tendrían dos dimensiones entre el punto en “x” o el punto en el eje “y”.

$a_r(x)$: representa el r-atributo o dimensión de la muestra x_e la cual está etiquetada, se tendría dos dimensiones entre el punto en “x” o el punto en el eje “y”.

x_s: es la muestra de la que se pretende predecir su etiqueta.

x: es el atributo o en este caso la coordenada en “x” o “y” de la que se quiere sacar la distancia.

En el modelo PNN se utilizará una función gaussiana como ventana. Por lo tanto, se tiene:

$$Pc(x_s, x) = \frac{1}{Nc \cdot \sigma\sqrt{2}} \sum_{i=1}^{Nc} e^{\frac{D(x_s, x_i)}{2\sigma^2}} \qquad (1.62)$$

Pc: probabilidad de que la muestra x_s, pertenezca a la clase c.

Nc: número de muestras de la clase c.

σ: representa el ancho de la curva. Para el caso de machine learning este valor se escoge de acuerdo con el que más convenga.

x_s: es la muestra de la que se pretende predecir su etiqueta.

x_i: es la i-muestra perteneciente a la clase c.

Se debe recordar que Pc nos indica la probabilidad de que la muestra x_s pertenezca a la clase c.

Finalmente, la etiqueta se escogería de la siguiente manera:

$$E_p = argmax(Pc), \qquad c = 1 \dots c \tag{1.63}$$

Donde:

E_p: corresponde a la etiqueta de la clase más probable según el ejercicio, c puede tomar valores desde 1 hasta c, que es el número máximo de clases de los datos.

1.3.3. CREACIÓN DE DATOS

Los datos sintéticos son datos artificiales fabricados por ordenadores, son datos creados según los parámetros de cada usuario para que estos se asemejen al mundo real. Estos pueden tener sus ventajas y desventajas como en la Figura 1.33.[33]

VENTAJAS:	DESVENTAJAS:
- No hay problemas con propiedad intelectual. - Son buenos para entender un concepto en particular.	- Se pueden introducir sesgos.

Figura 1.33 Ventajas y desventajas de creación de datos

Dentro de los problemas que enfrenta el machine learning, se tienen los datos los cuales serán utilizados para entrenar el algoritmo, estos normalmente al encontrarse representados gráficamente se puede observar una frontera de decisión y esta puede variar dependiendo del problema o del número de clases como se puede ver en la Figura 1.34 y Figura 1.35:

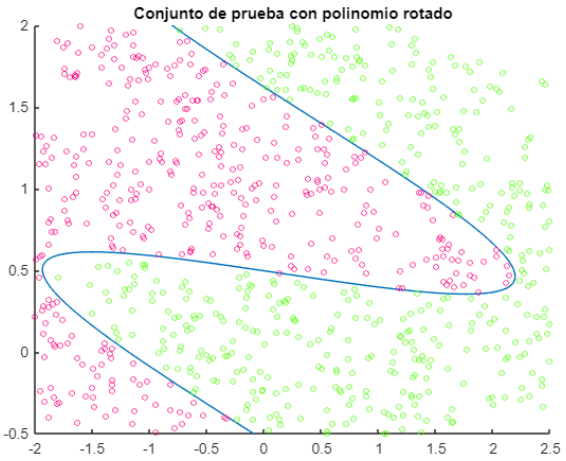


Figura 1.34 Conjunto de datos de 2 clases separados por una frontera de decisión.

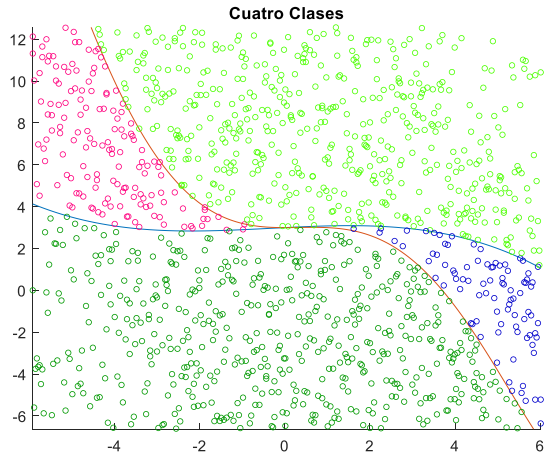


Figura 1.35 Conjunto de datos de 4 clases

Para la creación de datos artificiales tendremos en cuenta los siguientes apartados:

- Para 2 clases sin giro
- Para 2 clases con giro
- Para 4 clases

Al principio de este trabajo se presentaron algunas preguntas para evaluar algoritmos y para resolverlas necesitamos algunos datasets de entrenamiento y de prueba cuando se tienen 2 y 4 clases.

Para evaluar métricas de un algoritmo en base a su número de muestras de entrenamiento se usarán los algoritmos de la Tabla 1.1.

Tabla 1.1. Nombre de archivos de datos de entrenamiento con datasets de distintos números de muestras

Número de clases	Número de muestras de entrenamiento	Nombre de dataset en Matlab
2 clases	50 muestras (25 por cada clase)	Train2c50
	100 muestras (50 por cada clase)	Train2c100
	200 muestras (100 por cada clase)	Train2c200
	500 muestras (250 por cada clase)	Train2c500
	700 muestras (350 por cada clase)	Train2c700
4 clases	100 muestras (25 por cada clase)	Train4c100
	200 muestras (50 por cada clase)	Train4c200
	500 muestras (125 por cada clase)	Train4c500
	700 muestras (175 por cada clase)	Train4c700

Para evaluar a los algoritmos en función del grado de alejamiento se escogerá el dataset de entrenamiento con el que tenga mayor exactitud y sea menos propenso al sobreajuste.

Para evaluar los algoritmos en base al grado de asimetría en el dataset de entrenamiento se usarán los datasets de la Tabla 1.2.

Tabla 1.2. Nombre de archivos de datos de entrenamiento con datasets asimétricos

Número de clases	Número de muestras de entrenamiento	Nombre de dataset en Matlab
2 clases	300 muestras (100 muestras para clase 1 y 200 para clase2)	Train2c12
	400 muestras (100 muestras para clase 1 y 300 para clase2)	Train2c13
	500 muestras (100 muestras para clase 1 y 200 para clase2)	Train2c14

Para responder que tan bien responden a la distancia, al alejamiento y a la asimetría en los datos de entrenamiento se probarán con datasets de la Tabla 1.3.

Tabla 1.3 Nombres de archivos de datos de prueba

Número de clases	Número de muestras de entrenamiento	Nombre de dataset en Matlab
2 clases	200 muestras (100 por cada clase)	Test2c
	200 muestras cercanas a la frontera de decisión (100 por cada clase)	Test2cd1
	200 muestras a media distancia a la frontera de decisión (100 por cada clase)	Test2cd2
	200 muestras lejanas a la frontera de decisión (100 por cada clase)	Test2cd3
4 clases	400 muestras (100 por cada clase)	Train4c

Se debe recalcar que los conjuntos de datos de entrenamiento se guardarán en el archivo “Entrenamiento.mat” y los de prueba en “Prueba.mat”. El código explicado de la creación de cada archivo se hablará en la sección 2.1.3 y para acceder y crear conjuntos de datos diferentes se puede observar en ANEXO A.

1.3.3.1. Para 2 clases sin giro

Para la creación de una frontera polinomial simplemente se necesita de un polinomio, para este caso particular se utilizará el siguiente:

$$y = x^3 - 4x^2 + 2.5x - 5 \quad (1.64)$$

Para realizar el código, primero un se ingresa el número de puntos de la frontera, posteriormente se crea un vector que representará al eje de las abscisas, luego se crea el polinomio representado por la Ecuación (**1.64**), y finalmente se obtiene la gráfica de la Figura 1.36.

```
N = 100;
x = linspace(-0.5,3.25, N);
y = 1*x.^3-4*x.^2+2.5*x-5;
figure
plot(x, y, 'k')
title ('Frontera de decisión')
```

Resultado:

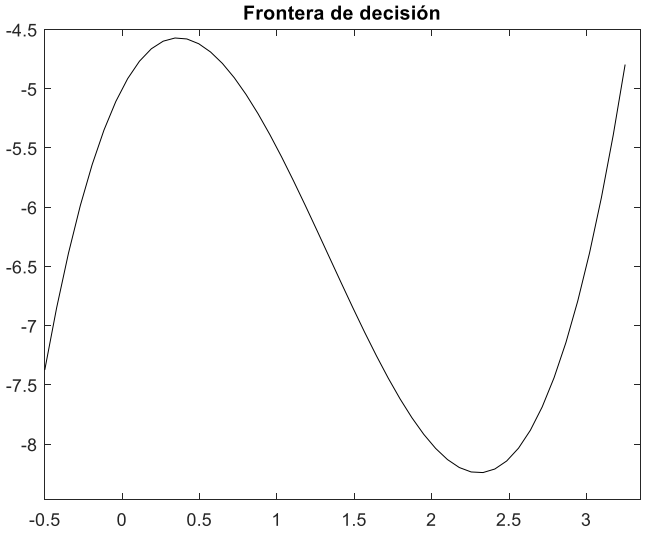


Figura 1.36 Frontera de decisión

1.3.3.2. Para 2 clases con giro

Una vez obtenido el polinomio se usa una matriz de giro para realizar la frontera de decisión.

Esta matriz de giro para el plano en 2 dimensiones por la siguiente expresión:

$$R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \quad (1.65)$$

Donde θ representa el ángulo en que rotan los datos creados de manera artificial.

Para obtener el nuevo polinomio rotado se debe incluir la siguiente ecuación:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (1.66)$$

Donde x' y y' corresponden a los nuevos ejes o en este caso los nuevos puntos rotados.

x y y pertenecen a los puntos a rotar.

A continuación, se puede ver el número de puntos que se utilizará para graficar la frontera, seguido se ve a la variable “alpha”, en ella se guardará el ángulo de giro.

```
N = 100; % Número de muestras
alpha = 65.495;     % Ángulo a rotar
```

Crearemos un vector que representará al eje de las abscisas, luego se crea el polinomio representado por la Ecuación (**1.64**), se agrupa todo en una matriz “p”.

```
x = linspace (-0.5,3.25,);
y = 1*x.^3-4*x.^2+2.5*x-5;
P = [x; y];
```

Para girar la frontera primero se crea la matriz de rotación “Mz” basada en la Ecuación (**1.65**), posteriormente se multiplica las dos matrices como se indica en Ecuación (**1.66**), y finalmente se obtiene la gráfica de la Figura 1.37.

```
% Matriz a usar
Mz = [cosd(alpha), -sind(alpha); ...
      sind(alpha), cosd(alpha)].
R = Mz*P.
figure
plot(R (1, :),R(2,:),'k');
title ('Frontera de decisión rotada')
```

Resultado:

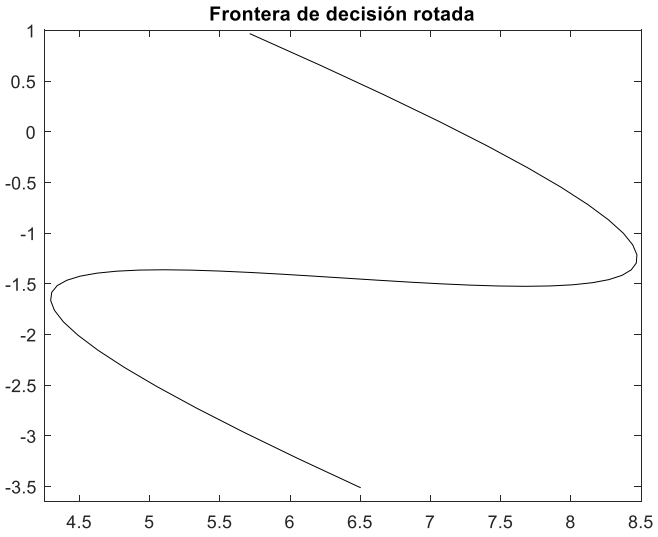


Figura 1.37 Frontera de decisión rotada

1.3.3.3. Para 4 clases

Cuando se trabajan con 4 clases se ha propuesto un conjunto de datos dividido por 2 fronteras de decisión, de ellas se tiene los siguientes polinomios:

$$y_1 = -0.005x^3 - 0.0001x^3 + 3x + 0.003 \quad (1.67)$$

$$y_2 = 0.001x^5 - 0.00001x^4 - 0.1x^3 + 0.1x^2 - 0.1x + 1 \quad (1.68)$$

A continuación, se observa en el código la misma receta que se ha aplicado. Luego en las variables “y1” y “y2” de la Ecuación (**1.67**) y Ecuación (**1.68**), respectivamente. Finalmente, se procede a graficar las dos fronteras con sus respectivas etiquetas en el mismo orden. El resultado se puede apreciar en la Figura 1.38.

```
n = 1000;
x = linspace (-8,8, n);
y1 = -0.00005*x.^3-0.0001*x.^2+3*x+0.03;
y2 = 0.001*x.^5+0.001*x.^4-0.1*x.^3+0.1*x.^2-0.1*x+0.03;
figure
plot (x, y1,x,y2)
title ('Fronteras de decisión para 4 clases')
legend ('Frontera de decisión 1',' Frontera de decisión 2')
```

Resultado:

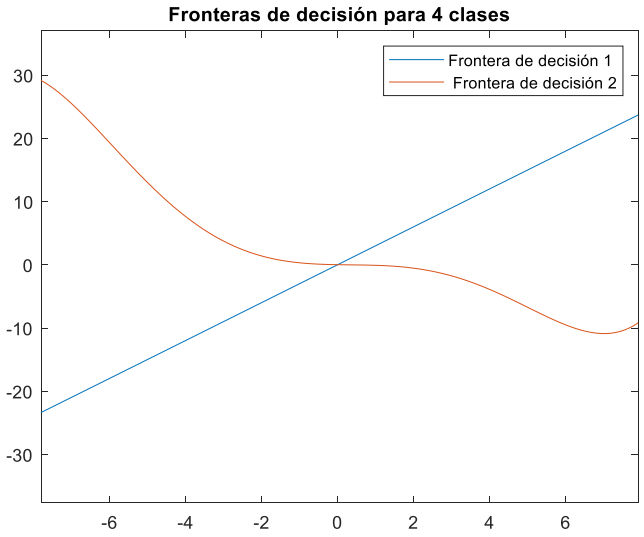


Figura 1.38 Frontera para 4 clases

1.3.1. MÉTRICAS DE ALGORITMOS DE CLASIFICACIÓN

1.3.1.1. Matriz de confusión

Una matriz de confusión es una matriz NxN que se utiliza para evaluar un modelo de clasificación donde N es el número de clases objetivo, esta matriz compara los valores objetivos reales con los predichos por el modelo del aprendizaje automático dando una idea del porcentaje de aciertos del modelo de clasificación y de los errores que se están cometiendo. Cabe resaltar que a partir de esta se pueden obtener otras métricas.

En la Figura 1.39 se puede apreciar los elementos de una matriz básica. Las columnas pertenecen a los valores reales de la variable objetivo, mientras que las filas representan a los valores predichos de la variable objetivo.

Figura 1.39 Representación de una matriz de confusión.

Para que la explicación quede clara lo explicaremos con un ejemplo de Wikipedia. Como se puede observar en la Figura 1.40, se hay tres clases, una pertenece a gatos, otra a perros y finalmente a conejos. En la Figura 1.40 se puede ver que hay 8 gatos, pero clasifica mal a 3 de ellos y los predijo como perros. Mientras que hay 6 perros, pero 2 de estos fueron clasificados como gatos, 1 como conejo y 3 fueron clasificados correctamente. Finalmente, se hallan 13 conejos entres los cuales 2 de ellos fueron clasificados como perros mientras que los 11 restantes fueron clasificados correctamente. Se puede concluir, que el clasificador funciona mejor con conejos ya que estos fueron en su mayoría clasificados correctamente, mientras que con los perros tiene más dificultades ya que la mitad de los perros pudo clasificar sin errores.[34]

		Valor Predicho		
		Gato	**Perro**	**Conejo**
Valor Real	**Gato**	5	3	0
	Perro	2	3	1
	Conejo	0	2	11

Figura 1.40 Ejemplo de una matriz de confusión. [35]

Para ejemplificar los siguientes conceptos se utilizará un ejemplo en el que un clasificador se dedica a predecir si una persona tiene cáncer o no.

Verdadero Positivo (TP)

Es el valor que ha predicho el modelo que coincide con el real. Significa que el valor real fue positivo y el modelo predice un valor positivo. Por ejemplo, cuando el clasificador predice que una persona sufre de cáncer y la persona efectivamente padece de este mal.

Verdadero negativo (TN)

El valor predicho coincide con el valor real. Significa que el valor real fue negativo y el modelo predice un valor negativo. Por ejemplo, el clasificador predice que la persona no tiene cáncer y en efecto la persona no lo padece.

Falso positivo (FP)

El valor predicho Falso significa que el valor real fue negativo pero el modelo predice un valor positivo. Por ejemplo, cuando el clasificador predice que la persona tiene cáncer, pero en realidad esta no padece de la enfermedad.

Falso negativo (FN)

El valor predicho fue falso significa que el valor real fue positivo, pero el modelo predijo un valor negativo. Por ejemplo, el clasificador predice que la persona no tiene cáncer, pero en realidad está si padece de la enfermedad.

1.3.1.2. Precisión

La precisión se refiere a los casos predichos correctamente que resultan ser positivos. Es una manera de medir la calidad del modelo. Ejemplo, dentro de todos los casos en el que el clasificador ha predicho que las personas están enfermas, qué porcentaje de estos en realidad tienen la enfermedad.

$$Precisión = \frac{TP}{TP + FP} \quad (1.69)$$

1.3.1.3. Exactitud

La exactitud mide el porcentaje de todos los casos en los que se ha acertado por consiguiente es una de las métricas más usadas. Ejemplo, dentro de todos los casos cuantos ha predicho el modelo correctamente tanto los que el clasificador ha dicho que están enfermos y en efecto lo están, como los que no están enfermos y el clasificador ha acertado. El problema en esta métrica se da cuando las clases no tienen el mismo número de elementos. Un ejemplo de exactitud se puede ver en la Figura 1.41.

$$Exactitud = \frac{TP + TN}{TP + FP + TN + FN} \quad (1.70)$$

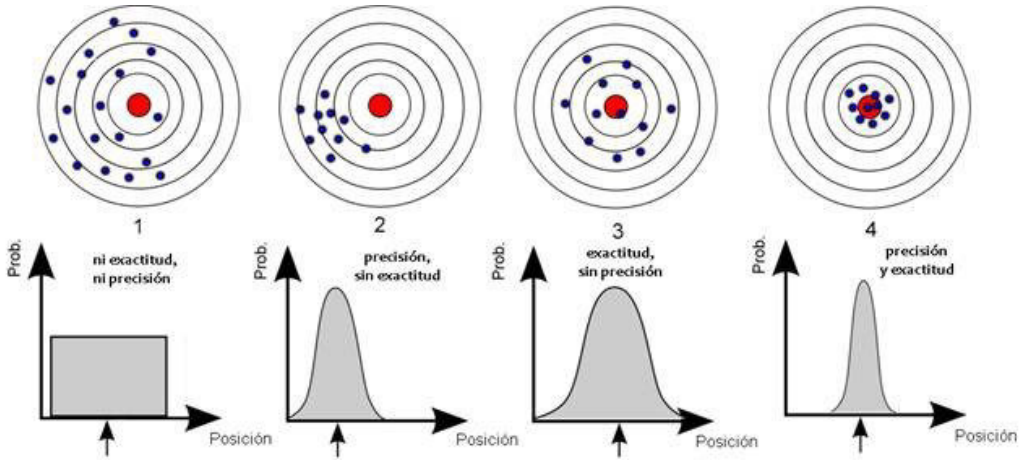


Figura 1.41 Diferencias entre precisión y exactitud.

Para las siguientes métricas se utilizará la Figura 1.42 para una explicación más intuitiva.

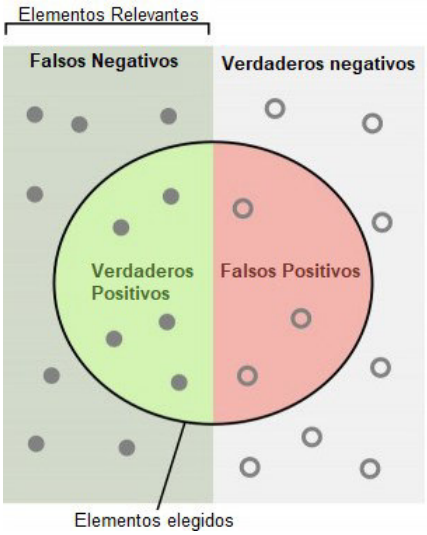


Figura 1.42 Representación de verdaderos positivos, falsos negativos, verdaderos negativos y falsos positivos.[35]

1.3.1.4. Sensibilidad

Esta métrica nos dice cuántos casos positivos reales se pueden predecir correctamente con este modelo. Lo que significa que nos va a avisar sobre la cantidad que el modelo es capaz de identificar. Ejemplo, se refiere a todos los casos que el clasificador que detectó como enfermos dentro de todos los enfermos, significa que es la capacidad que tiene de poder detectar correctamente la enfermedad de los que si están enfermos.

$$Sensibilidad = \frac{TP}{TP + FN} \qquad (1.71)$$

En la Figura 1.43 los elementos relevantes son aquellos pacientes que de verdad tienen cáncer, entonces la pregunta es ¿Qué porcentaje de pacientes que de verdad tienen cáncer puede clasificar correctamente el algoritmo?

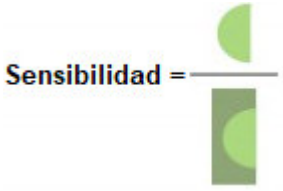


Figura 1.43 Sensibilidad de manera gráfica. [35]

La respuesta a la anterior pregunta nos da la sensibilidad, y en la figura anterior se ve que esta se saca dividiendo los verdaderos positivos para los verdaderos positivos más los falsos negativos. Esta métrica nos dice que tan bien funciona el clasificador detectando los verdaderos positivos o clase 1.

1.3.1.5. Especificidad

Es el número de muestras identificadas correctamente como negativas fuera del total de negativos. Por ejemplo, la métrica nos da indica la capacidad que tiene el clasificador de poder detectar a las personas sanas dentro de todas las que en realidad están sanas. [8]

$$Especificidad = \frac{TP}{TP + FP} \quad (1.72)$$

En la Figura 1.44, son las personas que están en realidad sanas. Bajo esa premisa la pregunta es ¿Qué porcentaje de personas que clasificó el algoritmo están sanas?

Figura 1.44 Especificidad de manera gráfica. [35]

La respuesta a la anterior pregunta lo tiene la especificidad, debido a que esta nos muestra el porcentaje de valores negativos que verdad son negativos, que es capaz de detectar el clasificador.

1.3.1.6. Medida F1

Es un valor que combina las métricas de precisión y sensibilidad.

Para entender a esta métrica se tiene que tomar en cuenta 4 casos:

- Precisión y Sensibilidad altas: el modelo maneja bien esa clase
- Precisión alta y sensibilidad baja: el modelo no detecta la clase, pero es muy confiable cuando predice.
- Precisión baja y sensibilidad alta: el modelo detecta bien la clase, pero no predice muy bien a qué clase pertenece.
- Precisión y sensibilidad bajas: el modelo no puede predecir la clase.

$$F1 = 2 \cdot \frac{Precisión \cdot Sensibilidad}{Precisión + Sensibilidad} \quad (1.73)$$

1.3.1.7. Overfitting

En el Aprendizaje automático supervisado los modelos que se diseñan deben generalizar los datos para que el modelo realice predicciones correctamente en el futuro con datos que

no conoce.[36] normalmente cuando se tienen malos resultados al entrenar modelos de machine learning se deben al “overfitting”.

El “overfitting” o sobreajuste se produce cuando el modelo se ajusta demasiado bien a los datos de entrenamiento, y como resultado cuando se producen nuevos datos de prueba desconocidos este no es capaz de predecir correctamente datos de prueba desconocidos.[37]

En la Figura 1.45 se puede ver que, aunque se entrena el modelo con 10 razas diferentes no es capaz de reconocer la nueva muestra el modelo debido a que no es del mismo tipo que las muestras entrenadas.

Figura 1.45 Ejemplo sencillo de sobre entrenamiento. [38]

En la Figura 1.46 se puede apreciar un ejemplo claro de sobreajuste.

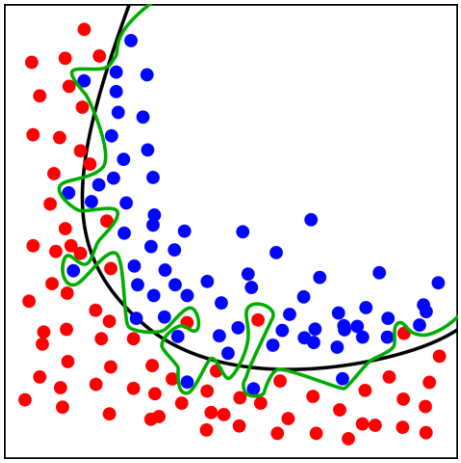


Figura 1.46 Sobreajuste de algoritmo. [39]

Identificación de overfitting.

A manera a de ejemplo, se analizará la Tabla 1.4 para detectar sobreajuste:

Tabla 1.4 Ejemplo sencillo de sobre entrenamiento

Modelo	Aciertos en el entrenamiento (%)	Aciertos en la prueba (%)
A	90	88
B	85	85
C	95	80

Los modelos A, B y C, son modelos de machine learning cualesquiera que nos servirán de ejemplo. Se puede ver que en el modelo “A” el porcentaje de aciertos en el entrenamiento es superior que el de prueba, pero no implica un sobreajuste porque normalmente los aciertos en el entrenamiento siempre van a ser superiores a los aciertos en los conjuntos de prueba. Entonces si se tiene la oportunidad de escoger entre el modelo “A” y “B”, se escoge el modelo “A”. Mientras que en el modelo “C”, se ve claramente un sobreajuste, debido a que el porcentaje de aciertos en el entrenamiento es muy superior al conjunto de datos de prueba.[12]

¿La complejidad de la red depende de la cantidad de datos?

La complejidad de la red no depende del tamaño del conjunto de datos, depende del número de entradas y de salidas de la red.

¿Cómo prevenir el sobreajuste?

La mejor manera de evitar el overfitting es seguir los procedimientos recomendados:

- Usar más datos de entrenamiento.

Es la forma más sencilla de mitigar este efecto, debido a que es más difícil para el modelo memorizar patrones exactos cuando se tienen más datos, naturalmente se ve obligado a obtener soluciones más flexibles a la hora de clasificar.[12]

- Usar menos características.

Puede ayudar al sobreajuste debido a que no permite que el modelo memorice demasiados campos con patrones específicos, por lo tanto, se debe ver qué características no son tan indispensables como para mantener la misma precisión a pesar de que estas no estén consideradas en el modelo. [12]

- Evitar pérdidas de destino.

La pérdida de destino se produce cuando el modelo “hace trampa”, significa que este accede a datos que no debería tener a la hora del entrenamiento.

- Identificar modelos no equilibrados.[12]

Se produce cuando un modelo es entrenado con un dataset en el que una de sus clases tiene más elementos que las otras, si se entrena con un dataset desequilibrado el modelo tendrá un sesgo hacia una clase.[12]

- Limitar la complejidad del modelo

Se da cuando un modelo como el de una red neuronal tiene muchas capas y neuronas, estas pueden ser innecesarias al entrenar un modelo, por lo tanto, se debe tratar de mantener un modelo que se adapte a este caso.[12]

1.3.1.8. Matriz de confusión en Matlab

En Matlab se tiene una función que nos permite graficar la matriz de confusión, a continuación, presentaremos un bloque de código que sirve para explicar cómo funciona.

Se puede observar que se asigna la variable "cm" a esta función para que se comporte como un objeto y poder modificar sus propiedades, en la función llamada "`confusionchart`" se tiene 6 entradas:

verdadero: corresponde a los valores reales que se quiere comparar.

predicho: son los valores predichos que salen del clasificador.

RowSummary: una opción que nos presenta un resumen de las filas.

row-normalized: nos muestra el número de observaciones clasificadas en porcentaje, en este caso la cantidad de aciertos o errores de los valores considerados como reales o verdaderos.

ColumnSummary: una opción que nos presenta un resumen de las columnas.

column-normalized: nos muestra el número de observaciones clasificadas en porcentaje, en este caso la cantidad de aciertos o errores de los valores predichos por el algoritmo.

Posteriormente se modifican las propiedades pertenecientes al título, etiqueta del eje "x" y etiqueta del eje "y", para que el lector pueda discernir entre los valores verdaderos y predichos.

```
cm = confusionchart (verdadero, predicho, 'RowSummary', 'row-normalized',
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
```

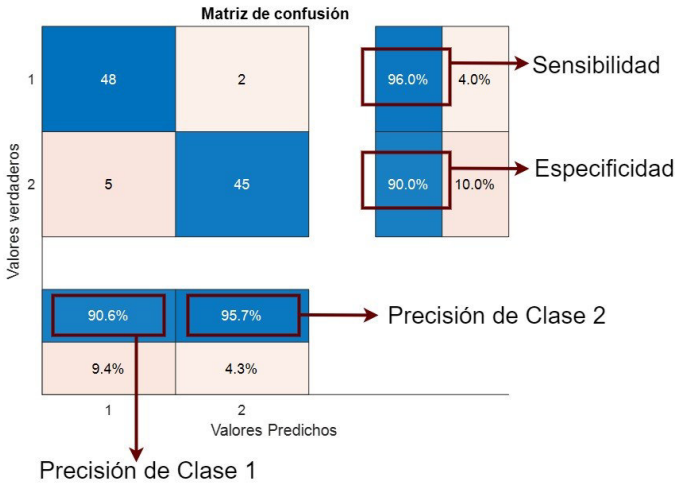


Figura 1.47 Ejemplo de matriz de confusión en Matlab

En la Figura 1.47, se puede observar que el resumen en porcentaje de la primera fila corresponde a la sensibilidad, y el de la segunda fila corresponde a la especificidad, pero En la misma Figura también se puede observar que el resumen de las columnas expresado en porcentaje corresponde a la precisión de cada clase.

2. METODOLOGÍA

Para la realización de este estudio comparativo se ha manejado la siguiente metodología:

Fase teórica: Se hará un breve análisis sobre el estado del arte respecto a los algoritmos de clasificación supervisada. Posteriormente, analizará los fundamentos para implementar datos artificiales. Finalmente se estudiarán los conceptos básicos de, al menos, los siguientes algoritmos: Support Vector Machine, Discriminante lineal, Decisión Tree, K-Near Neighbor, Multi Layer Perceptron.

Fase de diseño, análisis o implementación metodológica: Se instalará Matlab 2020 con las herramientas necesarias para comparar algoritmos de clasificación y redes neuronales. Se explorará la herramienta “Classification Learner”, que se encuentra en la Figura 2.1, entre los que se encuentran algoritmos como Support Machine vector, Decision Tree, Naive Bayes, K-Near Neighbor, Linear Discriminant Analysis, posteriormente se probará “Neural Network Start”, como se indica en la Figura 2.2, para probar la Multi Layer Perceptron. El objetivo de usar las herramientas anteriormente mencionadas es tener una base de partida para los algoritmos a implementar.

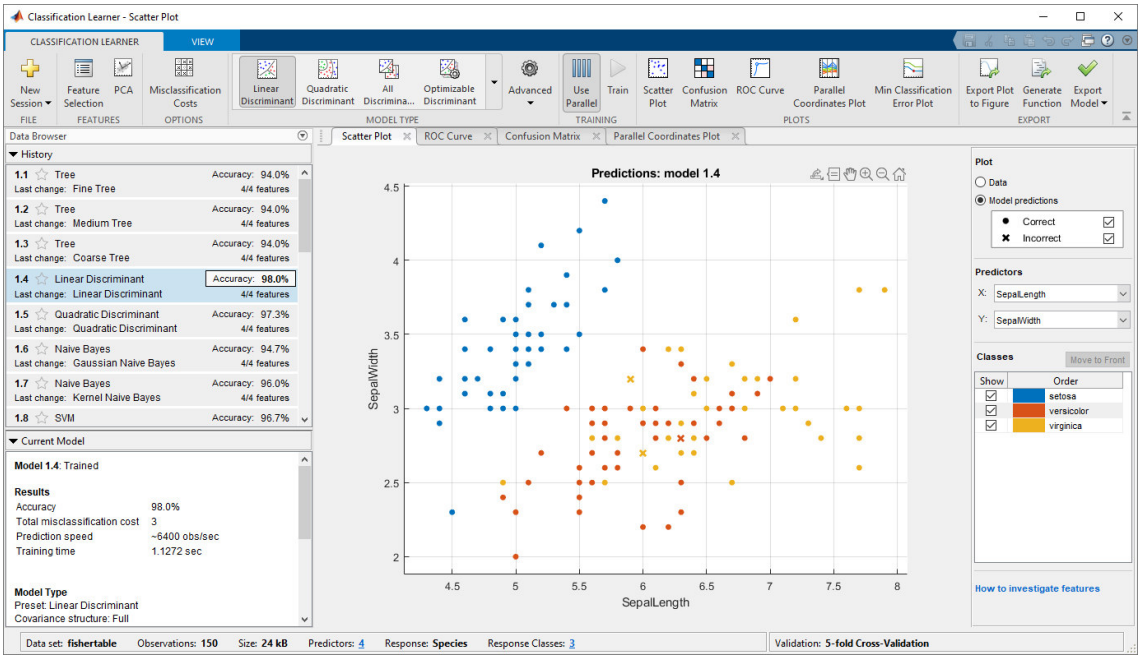


Figura 2.1 Ventana de la herramienta “Classification Learner”.

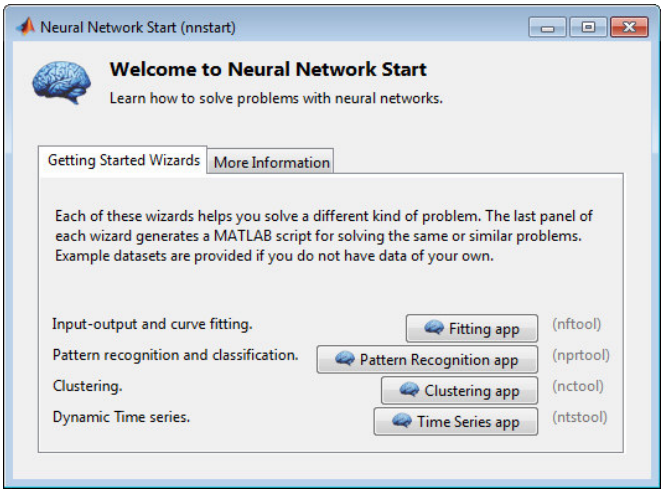


Figura 2.2 Ventana de inicio de la herramienta "Neural Network Start"

Fase de simulación y/o implementación: Se crean distintos grupos de datos artificiales con diferentes características que son el conjunto de entrenamiento de los clasificadores permitiendo determinar la exactitud bajo diferentes condiciones. Se implementan códigos propios en la herramienta de Matlab 2020 de los algoritmos antes mencionados.

Fase de validación / análisis de resultados/ pruebas de funcionamiento: Se entrenan los algoritmos implementados mencionados en el anterior apartado con los datos artificiales creados para comparar los resultados entre clasificadores y determinar cuál de ellos obtiene mejores resultados.

2.1. CREACIÓN DE DATOS

2.1.1. CONJUNTOS DE DATOS DE 2 CLASES

A continuación, se va a presentar la realización de los conjuntos de entrenamiento que serán usados para el entrenamiento de los algoritmos.

Esta parte del código estará contenida en una función llamada "generacion2clases", esta será muy utilizada cuando se tienen dos valores "dclase1" y "dclase2" que representan al número de datos por clase que se tendrán en los datasets.

La función tiene como salidas:

Data: es una matriz con 3 columnas en la primera el eje "x", en la segunda el eje "y" y en la tercera la etiqueta de cada muestra.

Frontera que se refiere a los datos necesarios para crear una frontera de decisión.

En las entradas de la función tenemos:

N: que corresponde a la cantidad de datos puntos que se pueden crear.

Val1: la distancia a la que el punto más cercano de la frontera de decisión puede estar, se debe reaclcar que solo se reciben números enteros.

Val2: la distancia a la que el punto más lejano puede estar se debe recalcar que solo se reciben números enteros.

dclase1: número de muestras que se quiere crear de la clase 1.

dclase2: número de muestras que se quiere crear de la clase 2.

```
%% Función de generación de Datos
function [data,Frontera]=generacion2clases(N, val1, val2, dclase1,
dclase2)
% N: número de muestras
% val1: distancia mínima de muestra a frontera de decisión
% val2: distancia máxima de muestra a frontera de decisión
% dclase1: datos finales de clase 1
% dclase2: dato finales clase 2
```

En las siguientes líneas se puede observar el mismo código de la sección 1.3.3.2, con la diferencia que aparece una nueva variable llamada “alpha”, en esta variable se guarda el valor del ángulo con el que va a rotar la matriz.

```
x = linspace(-0.5,3.25,N/2);          % valores eje x
y = 1*x.^3-4*x.^2+2.5*x-5;            % polinomio
da1 = y+0.025*randi ([val1, val2],1,N/2);%datos aleatorios clase 1
da2 = y-0.025*randi ([val1, val2],1,N/2);%datos aleatorios clase 2
%% Matriz y datos a usar
P = [x; y];    % vector que contiene punto a rotar
c1 = [x; da1]; % matriz clase 1 a rotar
c2 = [x; da2]; % matriz clase 2 a rotar
```

En las siguientes líneas se puede observar a la variable “Mz”, en esta variable se guarda la matriz de giro. A continuación, se puede ver que la matriz de giro es usada como en la Ecuación (**1.64**) para determinar los, nuevos puntos de los datos y la frontera de decisión.

```
% Matriz a usar
Mz = [cosd(alpha), -sind(alpha); ...
      sind(alpha), cosd(alpha)];
%% Operaciones
R = Mz*P;   % datos de función rotados
Rc1 = Mz*c1; % datos de clase 1 rotados
Rc2 = Mz*c2; % datos de clase 2 rotados
```

En las siguientes líneas se puede observar cómo se etiquetan los nuevos datos y se asigna la variable “frontera” los datos que corresponden a la misma frontera de decisión.

```
% Para datos rotados
clase1 = [Rc1.' zeros(N/2,1)];
```

```
clase2 = [Rc2.' ones(N/2,1)];
Frontera = R';
```

En las siguientes líneas se puede observar la asignación del conjunto de datos, siempre y cuando se tengan menos de 4 elementos en la entrada de la función, si no pasa eso se procede a volver a asignar la variable “data”, las muestras con el número de estas requerida.

```
if nargin < 4
    data = [clase1; clase2].
else
```

En las siguientes líneas se puede como se precarga las variables que serán parte de los datos que serán usados para entrenar o probar las redes.

```
    % Vectores donde se almacenarán los datos finales
    nVc1 = zeros (dclase1, size(clase1,2));
    nVc2 = zeros (dclase2, size(clase2,2));
```

En las siguientes líneas se inicializa a la variable “i” en cero, posteriormente se entra a un lazo que se termina cuando la variable antes mencionada alcanza el número de datos requeridos, posteriormente se guarda en la variable “j” un número aleatorio que forma parte de la clase 1, posteriormente se procede a asignar en el nuevo conjunto de datos de la clase 1 los valores escogidos al azar, posteriormente se procede a eliminar el elemento para que no vuelva a repetirse en el nuevo conjunto, se sigue con las clases correspondientes y finalmente el conjunto de entrenamiento es asignado de acuerdo al número de muestras requeridas por clase.

```
    % Escoger datos de la clase 1
    i=1
    while i < dclase1
        j = randi(length(clase1)-i).
        nVc1(i+1, :) = clase1(j, :);
        clase1(j, :) = [];
        i = i+1;
    end
    i=0;
    % Escoger datos de la clase 1
    while i < dclase2
        j = randi(length(clase2)-i).
        nVc2(i+1, :) = clase2(j,:);
        clase2(j, :) = [];
        i = i+1.
    end

    data = [nVc1; nVc2].
```

En las siguientes líneas se puede ver como se gráfica al conjunto de datos. Para asignar a una leyenda se asigna a una variable, para que sus propiedades sean manejadas

fácilmente y asignarle su correspondiente leyenda. El resultado se observa en la Figura 2.3.

```
figure
hold on
ele = 12.
colormap ([1 0 .5; % magenta
           1 .5 0; % anaranjado
           0 .6 0; % verde
           0 0 .8]); % azul
c1 =
scatter(datos(datos(:,3)==unicos(1),1),datos(datos(:,3)==unicos(1),2)...
    ,ele, datos(datos(:,3)==unicos(1),3));
c2 =
scatter(datos(datos(:,3)==unicos(2),1),datos(datos(:,3)==unicos(2),2)...
    ,ele,datos(datos(:,3)==unicos(2),3));
title('Dataset de entrenamiento con 2 clases 50 muestras')

title('Dataset con Frontera de decisión sin rotar')
hold off
```

Resultado:

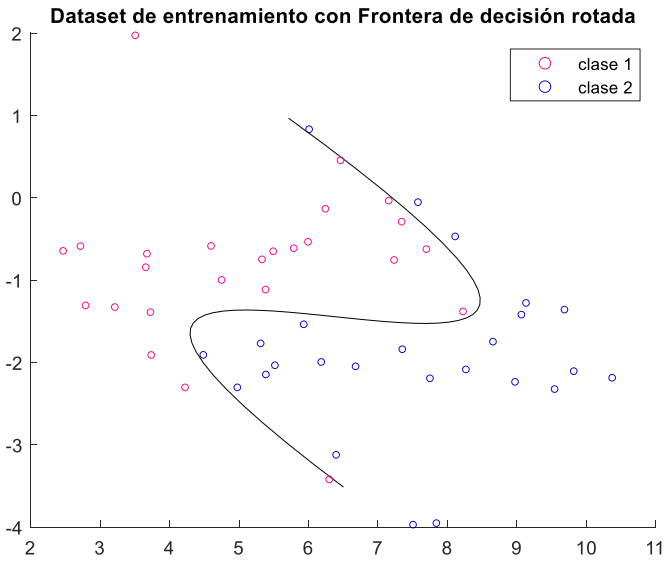


Figura 2.3 Dataset de entrenamiento con frontera rotada.

2.1.2. CONJUNTOS DE DATOS DE 4 CLASES

A continuación, presentaremos el código en Matlab empleado para este caso. Debido a que se usará distintas variaciones del mismo dataset este ha sido convertido a función con el nombre “generacion4clases”, consta de 5 entradas. La primera “N” consiste en un número de muestras que se usará en el dataset para crear los datos según número de datos que se necesitará por cada clase, para “c1”, “c2”, “c3” y “c4” pertenecen al número de muestras por clase que se quiere en el dataset. De salidas se tendrán las variables

“data” que contiene el conjunto de datos, además se tiene las variables “Frontera1” y “Frontera2” que no siempre se usará pero que contienen la frontera de decisión.

En el siguiente código se puede ver que “N” es una variable que determina el número de muestras, luego se puede observar “p”, que es el máximo valor que toman las muestras al alejarse de la frontera de decisión respectiva. La variable “q” regula la distancia a la que se encontrarán los datos de la frontera de decisión. Finalmente se tiene el mismo vector “x”, posteriormente usando las Ecuaciones (**1.67**) y (**1.68**) se forman las fronteras de decisión. Finalmente, estas últimas se agrupan ya que serán salidas de la función.

La función tiene como salidas:

Data: es una matriz con 3 columnas en la primera el eje “x”, en la segunda el eje “y” y en la tercera la etiqueta de cada muestra.

Frontera1: Frontera de decisión 1.

Frontera2: Frontera de decisión 2.

En las entradas de la función tenemos:

N: corresponde a la cantidad de datos que se pueden crear.

c1: número de muestras que se quiere crear de la clase 1.

c2: número de muestras que se quiere crear de la clase 2.

c3: número de muestras que se quiere crear de la clase 3.

c4: número de muestras que se quiere crear de la clase 4.

```
function [data, Frontera1, Frontera2] = generacion4clases(N, c1, c2, c3,
c4)

%Cuatro Clases Dos Polinomios
%%  Ingreso de datos
% N: Número de datos
% data: datos de salida
% Frontera1: Frontera de decisión de Primer polinomio
% Frontera2: Frontera de decisión de Segundo polinomio
% c1: número de muestras de clase 1
% c2: número de muestras de clase 2
% c3: número de muestras de clase 3
% c3: número de muestras de clase 4
```

En el siguiente código se puede ver que “N” es una variable que determina el número de muestras, luego se puede observar “p”, qué es el máximo valor que toman las muestras al alejarse de la frontera de decisión respectiva. La variable “q” regula la distancia a la que se encontrarán los datos de la frontera de decisión. Finalmente se tiene el mismo vector “x”,

posteriormente usando las Ecuaciones (**1.67**) y (**1.68**)se forman las fronteras de decisión. Finalmente, estas últimas se agrupan ya que serán salidas de la función.

```
% redondeo de datos
n = N*8;
if mod(n,8) ~= 0
    n = round(n/8) * 8;
end
% Separación máxima de puntos
p = 100;
% Tamaño de escala
q = 0.25;

x = linspace(-8,8,n/4);
%% creación de Frontera de decisión
y1 = -0.005*x.^3-0.0001*x.^2+3*x+0.03; %primer polinomio
y2 = 0.001*x.^5+0.001*x.^4-0.1*x.^3+0.1*x.^2-0.1*x+1; %segundo polinomio
Frontera1 = [x; y1]';
Frontera2 = [x; y2]';
```

En las siguientes líneas de código se crean las muestras alrededor de la frontera de decisión, posteriormente como se indica el título se asignan las muestras con sus respectivas clases. Luego se unen en una sola matriz para facilitar la gráfica.

```
c1_y1 = y1+q*randi([1,p],1,length(x)); % datos aleatorios clase 1
c2_y1 = y1-q*randi([1,p],1,length(x)); % datos aleatorios clase 2
c1_y2 = y2+q*randi([1,p],1,length(x)); % datos aleatorios clase 3
c2_y2 = y2-q*randi([1,p],1,length(x)); % datos aleatorios clase 4
% asignación de clases
clase1 = [x.' c1_y1.'  zeros(N/4,1)];
clase2 = [x.' c2_y1.'  ones(N/4,1)];
clase3 = [x.' c1_y2.'  2*ones(N/4,1)];
clase4 = [x.' c2_y2.'  3*ones(N/4,1)];
% creación de matriz que incluyen todos los datos
data = [clase1; clase2; clase3; clase4];
```

Si se grafican los datos como están hasta la anterior línea de código, se puede ver que no están clasificados correctamente, es por ellos que escribimos las siguientes líneas de código asegurando que los datos se queden en sus clases respectivas.

```
% Separación en clases
for i = 1:N/4
    if (c1_y1(i) > y1(i)) && (c1_y1(i) > y2(i))
        data(i,3) = 0;
    end
    if (c1_y2(i) > y1(i)) && (c1_y2(i) > y2(i))
        data(N/2+i,3) = 0;
    end
    if (c2_y1(i) < y1(i)) && (c2_y1(i) < y2(i))
        data(N/4+i,3) = 1;
    end
    if (c2_y2(i) < y1(i)) && (c2_y2(i) < y2(i))
```

```
        data(N*3/4+i,3) = 1;
    end
end


for i=1:N/8
    if (c1_y1(i) > y1(i)) && (c1_y1(i) < y2(i))
        data(i,3) = 2;
    end
    if (c2_y1(N/8+i) > y2(N/8+i)) && (c2_y1(N/8+i) < y1(N/8+i))
        data(N*3/8+i,3) = 3;
    end
    if (c2_y2(i) < y2(i)) && (c2_y2(i) > y1(i))
        data(N*3/4+i,3) = 2;
    end
    if (c1_y2(N/8+i) > y2(N/8+i)) && (c1_y2(N/8+i) < y1(N/8+i))
        data(N*5/8+i,3) = 3;
    end
end
```

Por último, se grafican los datos de la misma manera que se ha hecho con los anteriores datasets. El resultado se encuentra en la Figura 2.4.

```
figure
colormap([1 0.5 0;   % magenta
          0 .6 0;  % blue
          1 0 .5;% dark green
          0 0 .8]); % bright green
hold on
plot(Frontera1(:,1),Frontera1(:,2),'k')
plot(Frontera2(:,1),Frontera2(:,2),'k')
unicos =  unique(data(:,3));
c1 = scatter(data(data(:,3)==unicos(1),1),data(data(:,3)==unicos(1),2)...
    ,ele,data(data(:,3)==unicos(1),3));
c2 = scatter(data(data(:,3)==unicos(2),1),data(datos(:,3)==unicos(2),2)...
    ,ele,data(data(:,3)==unicos(2),3));
c3 = scatter(data(data(:,3)==unicos(3),1),data(data(:,3)==unicos(3),2)...
    ,ele,data(data(:,3)==unicos(3),3));
c4 = scatter(data(data(:,3)==unicos(4),1),data(data(:,3)==unicos(4),2)...
    ,ele,data(data(:,3)==unicos(4),3));
legend([c1,c2,c3,c4],{'clase 1','clase 2','clase 3','clase 4'})
title(['Dataset de prueba con 4 clases])
hold off
```

Resultados:

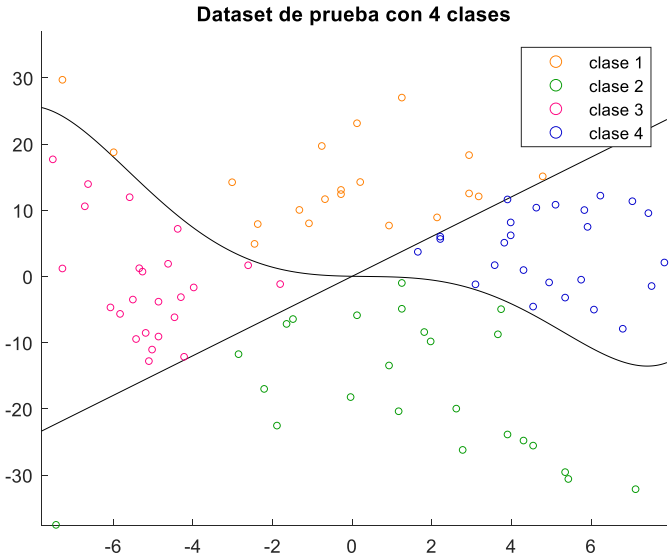


Figura 2.4 Dataset de prueba con 4 clases

2.1.3. PROGRAMA PARA CREACIÓN DE DATASETS DE ENTRENAMIENTO Y PRUEBA

En los apartados anteriores se muestran como se crean los conjuntos de entrenamiento, en este se verá el programa principal que llama a las funciones anteriores.

Para empezar, se limpian datos y variables que se encuentren en el código.

```
% Extracción de datos
clc, clear;
%% Creación de datos de entrenamiento
```

Se empieza llamando a la función “generacion2clases” para cada dataset nombrado de acuerdo con la Tabla 1.1. Nombre de archivos de datos de entrenamiento con datasets de distintos números de muestrasy Tabla 1.2. Se debe notar que en F2 se guarda la frontera de decisión, por si acaso se quiera graficar en un futuro.

En los siguientes datasets solo 3 entradas tiene cada función llamada donde se tienen valores aleatorios desde cercanos a los más lejanos de la frontera de decisión.

```
% Creación de datos de entrenamiento con distintos números de datos
train2c50 = generacion2clases(50, 1, 100);
[train2c100, F2] = generacion2clases(100, 1, 100);
train2c200 = generacion2clases(200, 1, 100);
train2c500 = generacion2clases(500, 1, 100);
train2c700 = generacion2clases(700, 1, 100);
```

En los siguientes datasets se generan 3000 puntos y de ellos se escogen algunas muestras en total de acuerdo con el grado de asimetría que se requiera.

```
% Con cantidad de datos desiguales relaciones 1:2, 1:3, 1:4, 2:3
%entrenamiento = generacionDatos(10000, 1, 100);
```

```
train2c12 = generacion2clases(3000, 1, 100, 100, 200);
train2c13 = generacion2clases(3000, 1, 100, 100, 300);
train2c14 = generacion2clases(3000, 1, 100, 100, 400);
```

Para 4 clases se llaman la función “generacion4clases” y con los valores en entradas de acuerdo con la Tabla 1.1 y Tabla 1.2.

```
% Cuatro clases entrenamiento con 100, 200, 500, 700
train4c100 = generacion4clases(100);
train4c200 = generacion4clases(200);
train4c500 = generacion4clases(500);
train4c700 = generacion4clases(700);
```

Para un dataset de pruebas se llama a la misma función que se utilizó para crear los datos de entrenamiento de 2 clases, pero en este caso se tendrán diferentes datos para probar.

```
%% Creación de datos de prueba
% Solo una clase y después otra
test2c = generacion2clases(700, 1, 100, 100, 100);
```

Para probar los algoritmos en base al grado de alejamiento se cambian los parámetros de la función que corresponden al alejamiento de la fontera de decisión.

```
% Creación de datos a distintas distancias de las fronteras de decisión
test2cd1 = generacion2clases(700, 1, 10, 100, 100);
test2cd2 = generacion2clases(700, 41, 50, 100, 100);
test2cd3 = generacion2clases(700, 121, 130, 100, 100);
```

Para un dataset de pruebas se llama a la misma función que se utilizó para crear los datos de entrenamiento de 4 clases, pero en este caso se tendrán diferentes datos para probar.

```
% Pruebas cuatro clases
test4c = generacion4clases(400);
```

Para cada uno los datasets de entrenamiento se guardan en el archivo “Entrenamiento.mat”.

```
filename = 'Entrenamiento.mat';
save(filename,'train2c50','train2c100','train2c200','train2c500','train2c
700', 'train2c12','train2c13','train2c14','train4c100','train4c200',...
'train4c500','train4c700');
```

Para cada uno los datasets de entrenamiento se guardan en el archivo “Prueba.mat”.

```
filename = 'Prueba.mat';
save(filename,'test2c','test2cd1','test2cd2','test2cd3','test2c12','test2
c13','test2c14','test4c','F2','F41','F42');
```

2.2. HERRAMIENTAS PROPIAS DE MATLAB

Matlab tiene varios caminos para implementar algoritmos de clasificación entre ellos están:

- Clasification Learner

- Neural Network Start

2.2.1. CLASSIFICATION LEARNER

Para utilizar diversos algoritmos de clasificación es una herramienta muy útil. Entre ellos se puede encontrar: DT, Análisis de discriminante, SVM, Regresión Logística, KNN, NB, conjuntos y redes neuronales. En esta sección se va a ver una pequeña introducción sobre el funcionamiento de esta aplicación de Matlab.

A continuación, se verán los pasos para acceder a la aplicación Classification learner.

Primer paso es situarse en la barra de herramientas de Matlab en APPS, y extender la pestaña principal, y escoger Classification Learner, como se puede observar en la Figura 2.5.

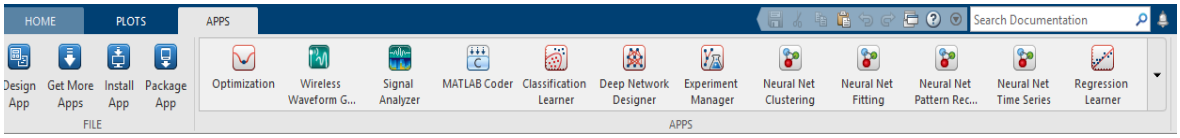


Figura 2.5 Localización de Classification Learner.

Una vez abierta la ventana de Classification Learner se le da clic en la pestaña de New session (Figura 2.6), donde se encontrará las opciones si se desea que los datos para entrenar sean del Workspace o de algún archivo. Para este caso primero insertamos el siguiente comando en el Worksapce:

```
fishertable = readtable('fisheriris.csv');
```

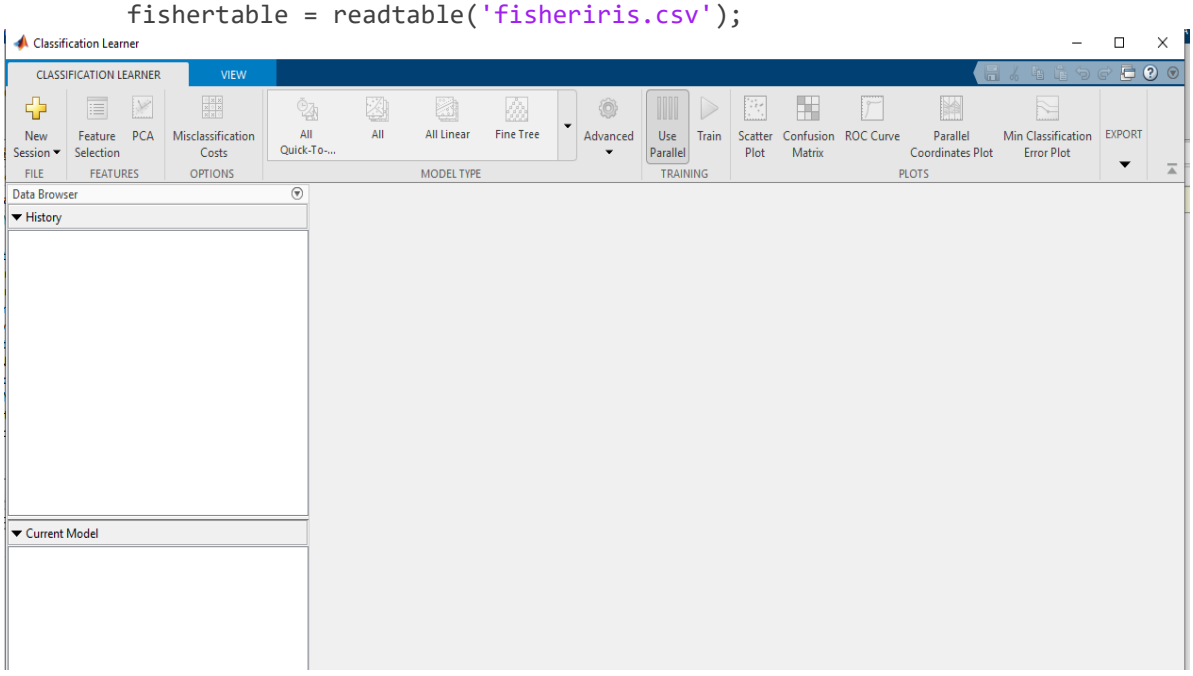


Figura 2.6 Ventana principal del Classsification Learner.

Se pueden encontrar más ejemplos, con sus comandos en el enlace [6].

Una vez que está en New Session, se eligen los datos que se utilizarán, cabe recalcar que normalmente recibe tablas, una vez que se eligen los parámetros adecuados se le da clic, en Start Session.

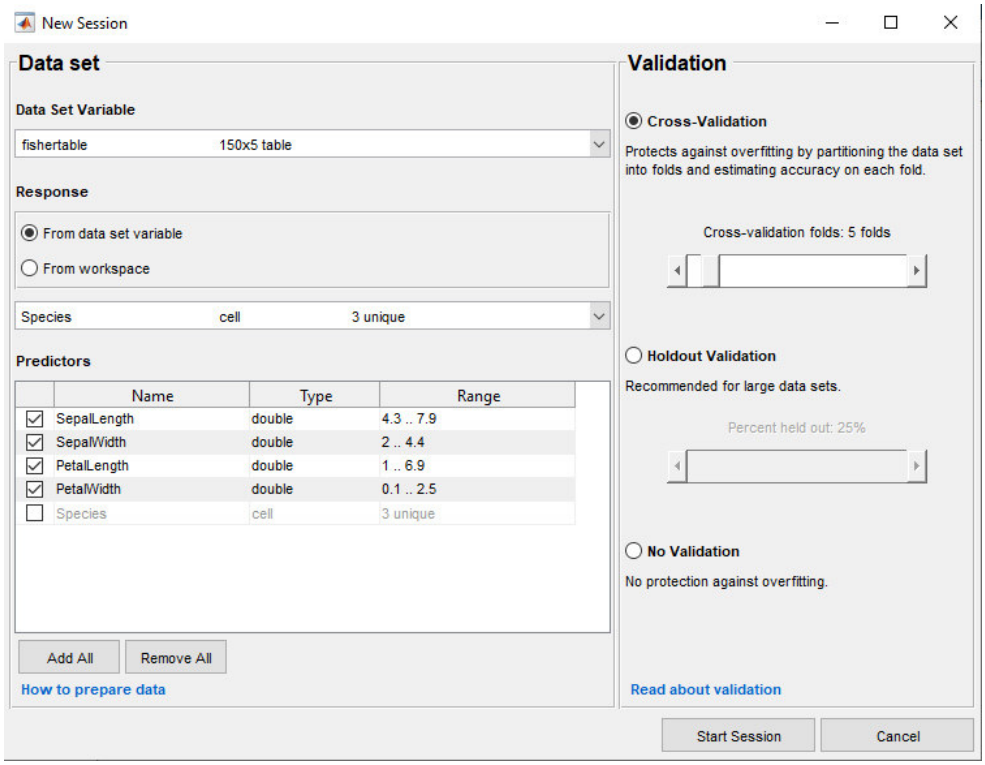


Figura 2.7 Ventana para cargar datos.

Como se indica en la Figura 2.7, se elige un método de validación para examinar la precisión predictiva de los modelos ajustados. La validación estima la exactitud del modelo en los nuevos datos en comparación con los datos de capacitación y lo ayuda a elegir el mejor modelo. La validación protege contra el sobreajuste.

Se puede elegir un método de validación entre ellos se tiene:

Validación cruzada: selecciona una cantidad de pliegues (o divisiones) para dividir el conjunto de datos con el control deslizante.

Si se elige k pliegues, entonces la aplicación:

Particiona los datos en k conjuntos disjuntos o pliegues

Para cada pliegue:

- Entrena un modelo utilizando las observaciones fuera de pliegue
- Evalúa la exactitud del modelo utilizando datos plegados
- Calcula el error de prueba promedio en todos los pliegues

Este método proporciona una buena estimación de la precisión predictiva del modelo final entrenado con todos los datos. Requiere ajustes múltiples, pero hace un uso eficiente de todos los datos, por lo que se recomienda para pequeños conjuntos de datos.

Validación de retención: selecciona un porcentaje de los datos para usar como un conjunto de prueba usando el control deslizante. La aplicación entrena un modelo en el conjunto de entrenamiento y evalúa su exactitud con el conjunto de prueba. El modelo final se entrena con el conjunto de datos completo.

Sin validación: sin protección contra el sobreajuste. La aplicación utiliza todos los datos para el entrenamiento y calcula la tasa de error en los mismos datos. Sin ningún dato de prueba, obtiene una estimación poco realista de la exactitud del modelo con los nuevos datos. Es decir, es probable que la precisión de la muestra de entrenamiento sea irrealmente alta y que la precisión predictiva sea menor.

Una vez que se tiene los datos cargados se escoge qué clasificador se desea en la sección de Model type, y se le da clic a Train para que empiece a entrenar. Presentando el resultado en la Figura 2.8.

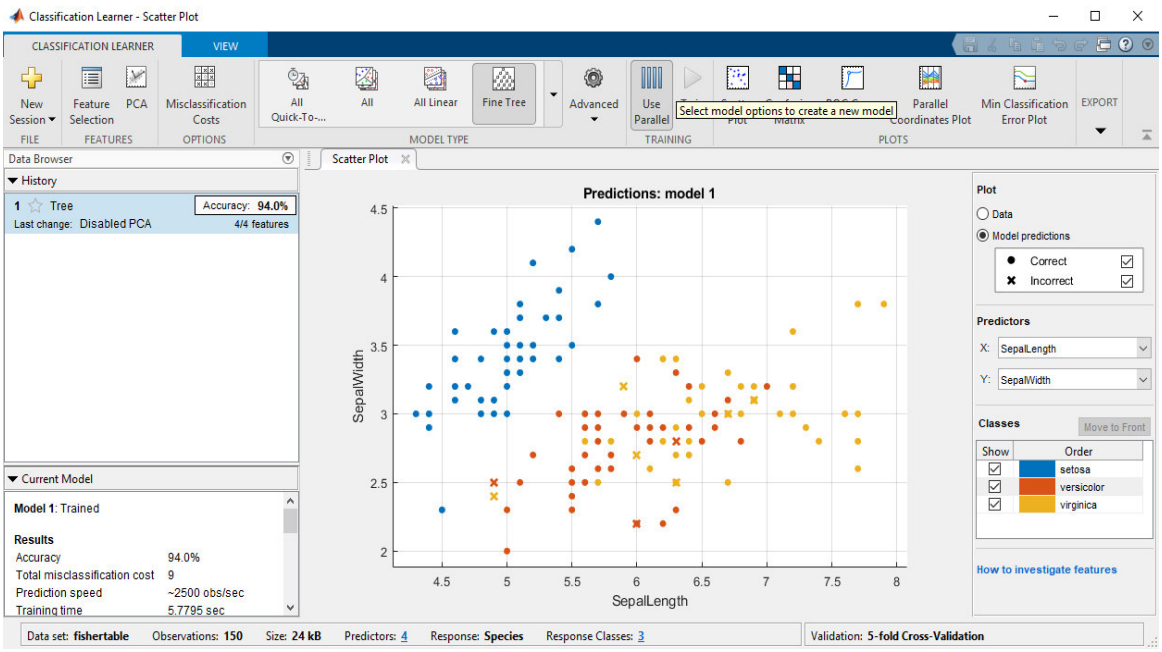


Figura 2.8 Entrenamiento de Datos.

Cuando se ha entrenado el clasificador se puede observar en la Figura 2.9 la matriz de Confusión y en la Figura 2.10, la curva ROC.

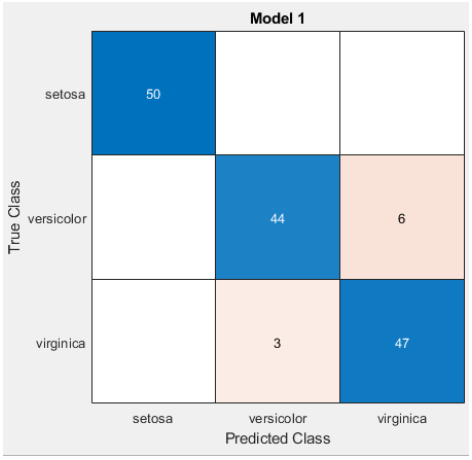


Figura 2.9 Matriz de confusión del entrenador.

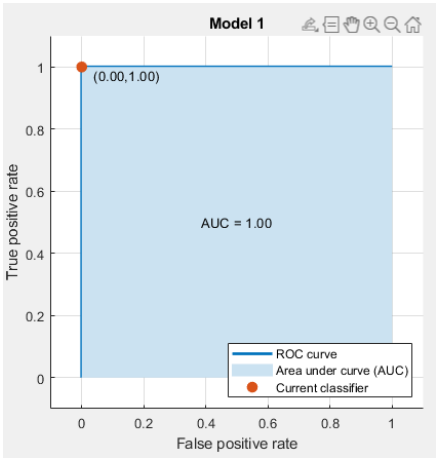


Figura 2.10 Curva de ROC.

Por último, se puede exportar la figura, una función, o exportar modelo de machine learning como se indica en la Figura 2.11.

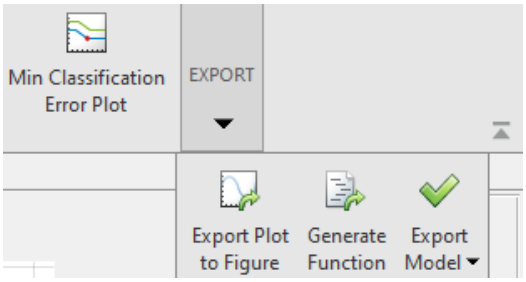


Figura 2.11 Exportación de Código.

Se puede generar código MATLAB para:

- Entrenar en grandes conjuntos de datos. Explore modelos en la aplicación entrenados en un subconjunto de sus datos, luego genere código para entrenar un modelo seleccionado en un conjunto de datos más grande.

- Crear scripts para modelos de entrenamiento sin necesidad de aprender la sintaxis de las diferentes funciones.
- Examinar el código para aprender a entrenar clasificadores mediante programación.
- Modificar el código para un análisis posterior, por ejemplo, para configurar opciones que no puede cambiar en la aplicación
- Repetir su análisis en diferentes datos y automatice la capacitación.

Entre los métodos que se pueden utilizar con esta herramienta se encuentran son siguientes:

SUPPORT VECTOR MACHINE

DECISION TREES

K-NEAREST-NEIGHBOR

NAIVE BAYES

2.2.2. NEURAL NETWORK START

En general, es mejor comenzar con la GUI y luego usar la GUI para generar automáticamente scripts de línea de comandos. Antes de usar cualquiera de los métodos, el primer paso es definir el problema seleccionando un conjunto de datos. Para ellos se dará una guía.

Tipear en el GUI de inicio el comando nnstart. Se presenta la venta de la Figura 2.12.

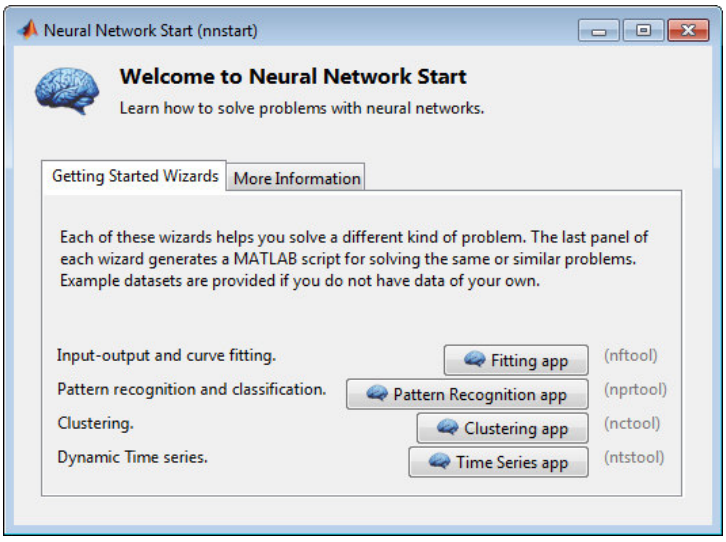


Figura 2.12 Ventana de inicio para Neural Network

Elegir la opción aplicación de reconocimiento de patrones, posteriormente se abrirá la ventana de la Figura 2.13.

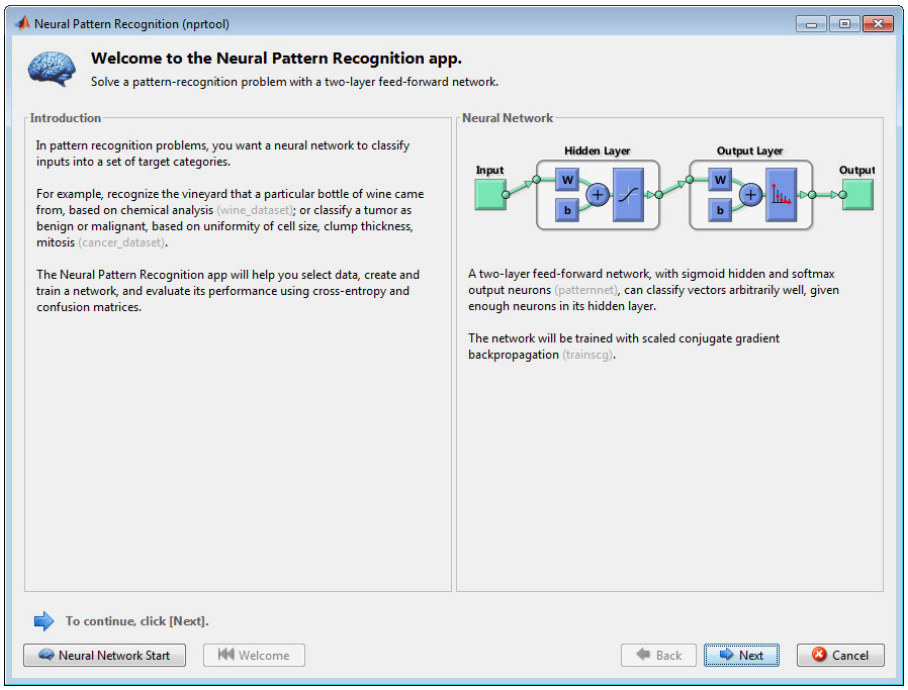


Figura 2.13 Ventana para comenzar.

Hacer clic en continuar, donde se abrirá la ventana de datos (Figura 2.14).

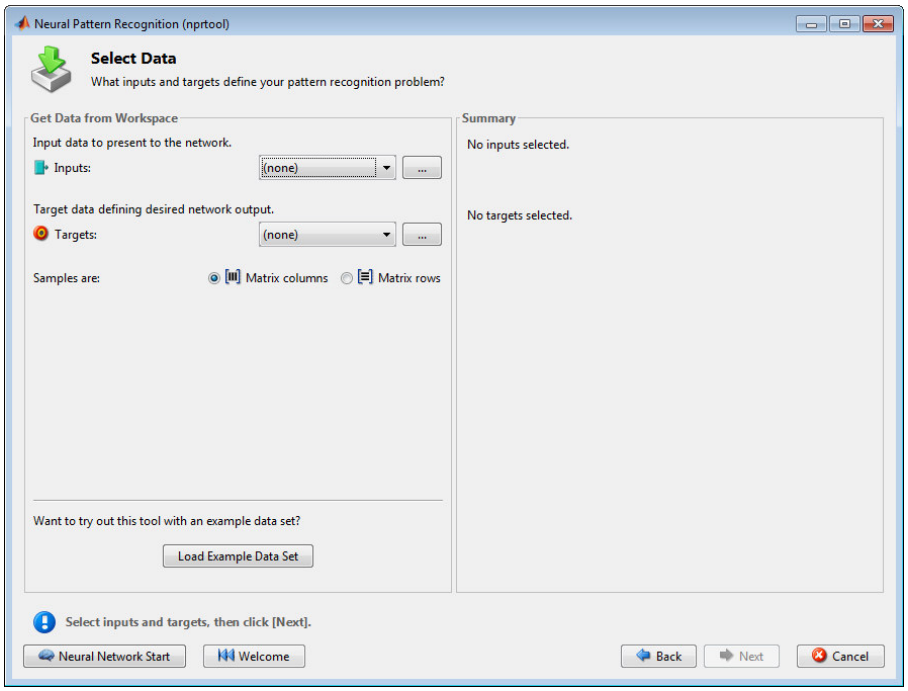


Figura 2.14 Ventana de selección de datos.

Como este es un ejemplo se puede poner en cargar datos, si es que tiene datos de entrada previamente cargados se elige como en la ventana de la Figura 2.15.

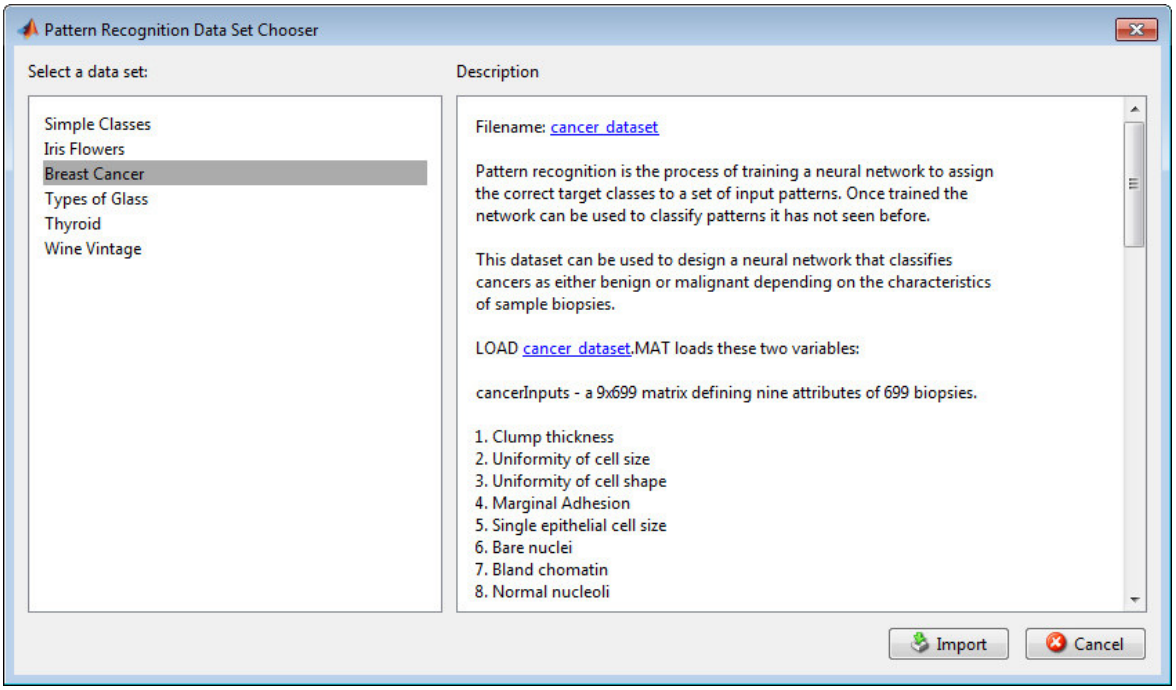


Figura 2.15 Ventana para cargar datos de ejemplos

Después de elegir los tipos de datos de ejemplo dar en importar o se puede dar en siguiente.

La red por lo general tendrá dos capas con una función sigmoidea, en la capa oculta y una función de transferencia softmax en la capa de salida. En la ventana de la Figura 2.16 se observa que el número predeterminado de neuronas ocultas se establece en 10. Es posible que desee volver y aumentar este número si la red no funciona tan bien como espera.

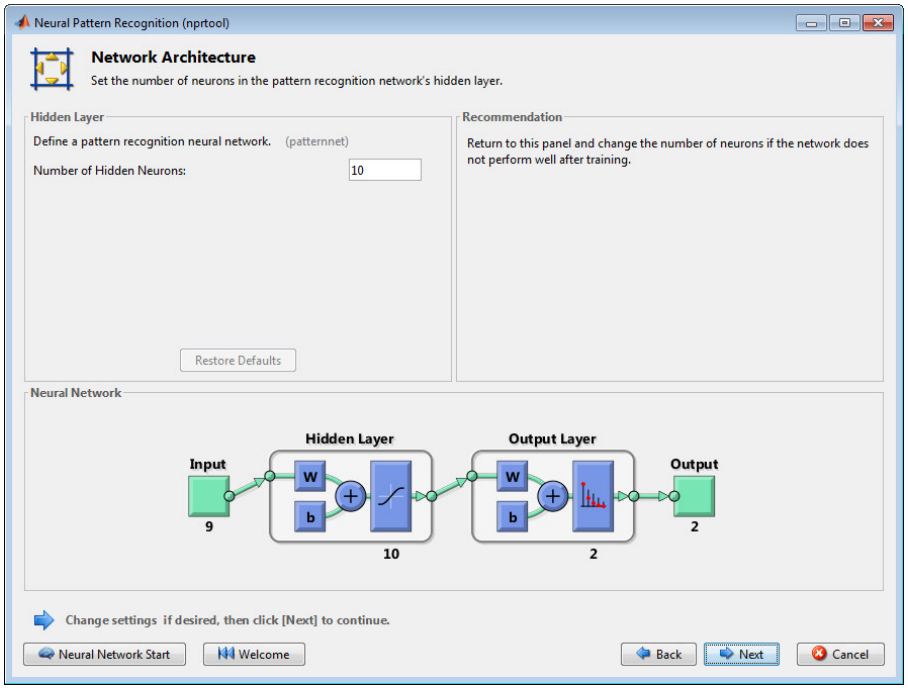


Figura 2.16 Ventana para elegir el número de neuronas.

Posteriormente se puede observar los resultados y entrenamiento de la red en la Figura 2.17.

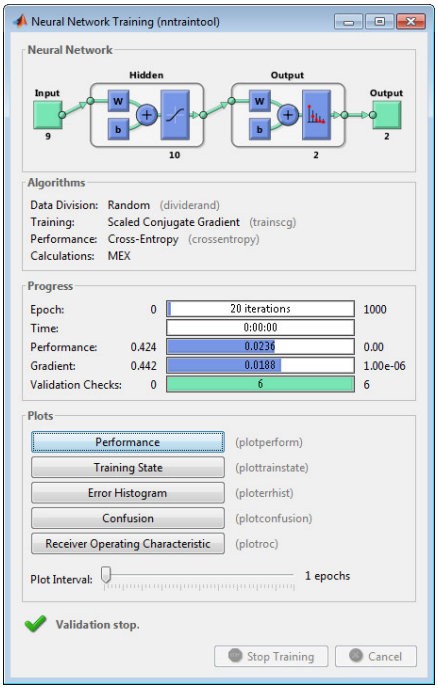


Figura 2.17 Ventana de entrenamiento.

La Figura 2.18 muestra la matriz de confusión para entrenamiento, pruebas, validación, y los tres tipos de datos combinados.

También se puede observar en la Figura 2.19, la curva ROC es un gráfico de la tasa positiva verdadera (sensibilidad) versus la tasa positiva falsa (especificidad 1) a medida que se varía el umbral.

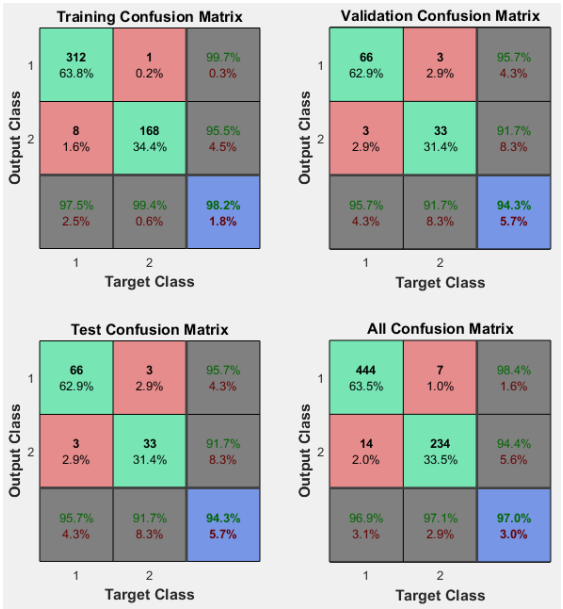


Figura 2.18 Ventana de entrenamiento de red Neuronal.

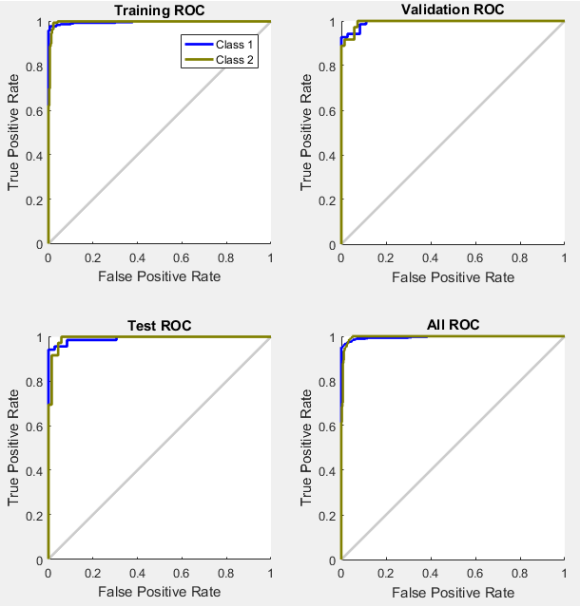


Figura 2.19 Figuras de ROC.

Si no se está satisfecho con los datos se puede entrenarlo nuevamente o aumentar el número de neuronas para obtener un conjunto de datos de entrenamiento más grande, como se indica en la Figura 2.20.

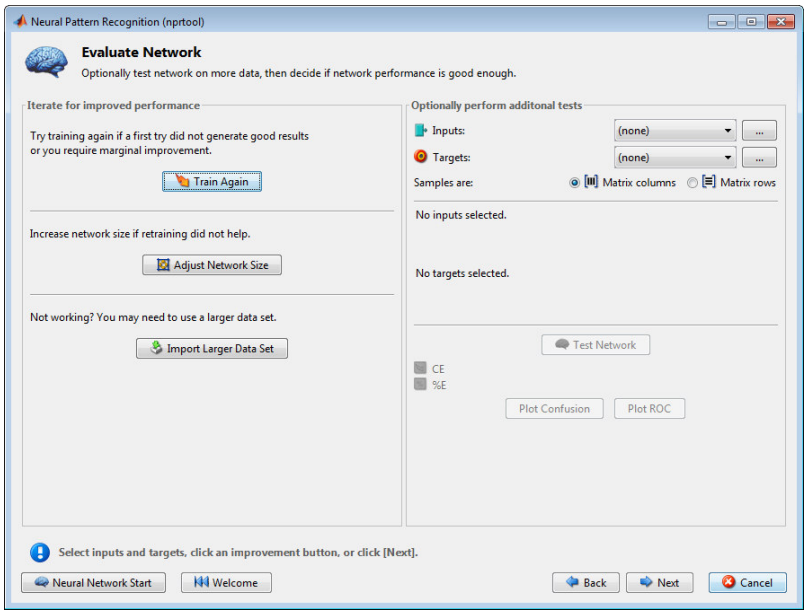


Figura 2.20 Ventana de evaluación de la red.

Al hacer clic en siguiente hay diversas opciones donde se puede usar el código de la fuente para hacer modificaciones (Figura 2.21).

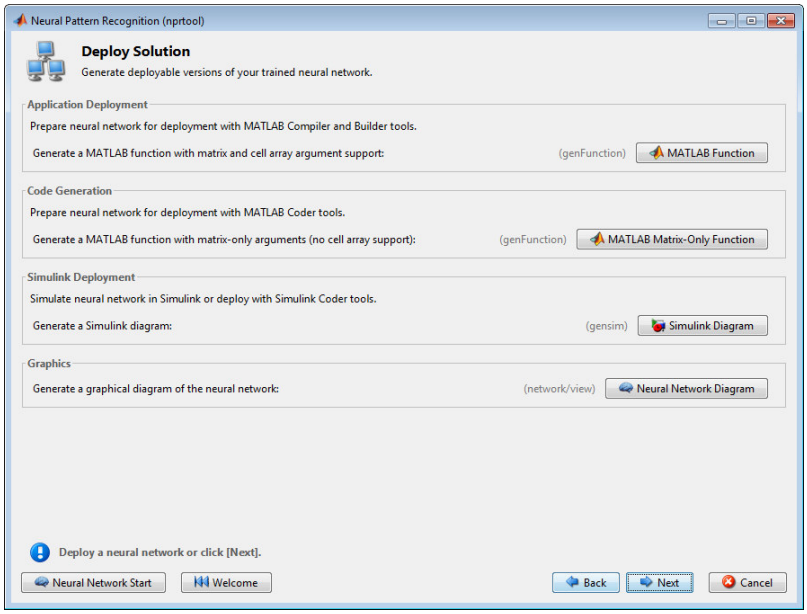


Figura 2.21 Ventana de exportación de código.

La forma más fácil de aprender a usar la funcionalidad de la línea de comandos de la caja de herramientas es generar secuencias de comandos a partir de las GUI y luego modificarlas para personalizar la capacitación de la red.

```
% Solve a Pattern Recognition Problem with a Neural Network
% Script generated by NPRTOOL
load ejem1.mat
inputs = RT';
targets = clases';

% Create a Pattern Recognition Network
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);

% Set up Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Train the Network
[net,tr] = train(net,inputs,targets);

% Test the Network
outputs = net(inputs);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)

% View the Network
view(net)

% Plots
% Uncomment these lines to enable various plots.
% figure, plotperform(tr)
% figure, plottrainstate(tr)
 figure, plotconfusion(targets,outputs)
% figure, ploterrhist(errors)
```

2.3. IMPLEMENTACIÓN DE ALGORITMOS DE MACHINE LEARNING

Para la creación de algoritmos se estructuró el código principal de cada algoritmo de la siguiente forma:

- Se ingresan los datos de entrenamiento o muestras con etiquetas conocidas.
- Se ingresan datos de pruebas que también contienen etiquetas conocidas, pero estas no son utilizadas para entrenar el algoritmo solo son para comparar los resultados de los algoritmos.
- Algoritmo: en esta sección se aplicará el algoritmo como tal.
- Resultados: en esta sección se usará varias funciones externas para publicar los resultados, matrices de confusión u otras medidas que se necesitan para evaluar los algoritmos.

Las funciones que generalmente se usará son:

- Función “Graficar”

2.3.1. FUNCIÓN GRAFICAR

La función “Graficar” va a ser usada en todos los algoritmos para clasificación para que la visualización de los datos de prueba sea lo más intuitiva posible. Esta función no tiene salidas ya que su único objetivo es graficar, pero consta de las siguientes entradas: “eti_Train”, “eti_Test”, “XY_Test”, “sal_malla” y “coordenadas”.

`eti_Train`: corresponde a las etiquetas que predijo el clasificador.

`eti_Test`: corresponde a las etiquetas reales.

`XY_Test`: son los datos que se planea graficar.

`sal_malla`: es el vector que contiene la etiqueta tentativa de cada malla.

`coordenadas`: corresponde a la malla para dar una idea de la frontera de decisión aproximada que maneja el clasificador.

En las siguientes líneas de código se inicializa la figura, y ponemos una condición que consiste en que, si la función tiene más de 3 argumentos de entrada, se grafica la malla, caso contrario se procede con el código normalmente. Luego se inicializa el tamaño de la muestra con la variable “ele” para que sea más visible, también se inicializa “n” que es grueso de las líneas de la muestra, estas variables facilitarán la visión de las muestras.

```
function Graficar(eti_Train, eti_Test, XY_Test, sal_malla, coordenadas)
figure
if nargin > 3
imagesc(coordenadas(:,1),coordenadas(:,2),sal_malla); colormap hsv ;
end
hold on
ele = 50;
n = 1.5;
```

Para algoritmos como arboles de decisión necesitamos otros colores para crear contraste, es por ellos que las siguientes líneas que están como comentario, por lo tanto, si las se quiere usar simplemente se debe quitar el símbolo de comentario para utilizarlas.

```
% colormap([1 0 0; % Rojo
%            1 .2 1; % amarillo
%            .1 .1 .5; % verde
%            0 1 0; % azul
%            .65 .75 1; % amarillo claro
%            .55 .85 1; % rojo claro
%            .75 .65 1; % verde claro
%            .85 .55 1]); % azul
```

En las siguientes líneas de código se observa que si se tiene más de 2 clases se procede un dataset de 4 clases. En “dcc1”, “dcc2”, “dcc3” y “dcc4” se almacenan las muestras que han sido predichas correctamente y se grafican a la vez correspondientes a la clase 1, clase 2, clase 3 y clase 4 respectivamente. Mientras que en las variables “dnc1”, “dnc2”, “dnc3” y “dnc4” se guardan las muestras predichas erróneamente por el algoritmo.

```
if length(unique(eti_Test)) > 2
    % muestras predichas correctamente de clase 1
    dcc1 = scatter(XY_Test(1, eti_Train == 0 & eti_Train == eti_Test),...
           XY_Test(2, eti_Train == 0 & eti_Train == eti_Test),...
           ele, eti_Test(eti_Train == 0 & eti_Train == eti_Test));
    % muestras predichas correctamente de clase 2
    dcc2 = scatter(XY_Test(1, eti_Train==1 & eti_Train == eti_Test),...
           XY_Test(2, eti_Train == 1 & eti_Train == eti_Test),...
           ele, eti_Test(eti_Train==1 & eti_Train == eti_Test));
    % muestras predichas correctamente de clase 3
    dcc3 = scatter(XY_Test(1, eti_Train == 2 & eti_Train == eti_Test),...
           XY_Test(2, eti_Train == 2 & eti_Train == eti_Test),...
           ele, eti_Test(eti_Train == 2 & eti_Train == eti_Test));
    % muestras predichas correctamente de clase 4
    dcc4 = scatter(XY_Test(1, eti_Train == 3 & eti_Train == eti_Test),...
           XY_Test(2, eti_Train == 3 & eti_Train == eti_Test),...
           ele, eti_Test(eti_Train == 3 & eti_Train == eti_Test));
    % muestras predichas incorrectamente de clase 1
    dnc1 = scatter(XY_Test(1,eti_Test==0 & eti_Train~=eti_Test),...
           XY_Test(2,eti_Test==0 & eti_Train~=eti_Test),...
           ele,eti_Test(eti_Test==0 & eti_Train~=eti_Test),'*');
    % muestras predichas incorrectamente de clase 2
    dnc2 = scatter(XY_Test(1,eti_Test==1 & eti_Train~=eti_Test),...
           XY_Test(2,eti_Test==1 & eti_Train~=eti_Test),...
           ele,eti_Test(eti_Test==1 & eti_Train~=eti_Test),'*');
    % muestras predichas incorrectamente de clase 3
    dnc3 = scatter(XY_Test(1,eti_Test==2 & eti_Train~=eti_Test),...
           XY_Test(2,eti_Test==2 & eti_Train~=eti_Test),...
           ele,eti_Test(eti_Test==2 & eti_Train~=eti_Test),'*');
    % muestras predichas incorrectamente de clase 4
    dnc4 = scatter(XY_Test(1,eti_Test==3 & eti_Train~=eti_Test),...
           XY_Test(2,eti_Test==3 & eti_Train~=eti_Test),...
           ele,eti_Test(eti_Test==3 & eti_Train~=eti_Test),'*');
```

En las siguientes líneas de código se establece el ancho de la línea a ser graficada de cada variable creada anteriormente.

```
    dcc1.LineWidth  = n; dcc2.LineWidth  = n;dcc3.LineWidth  = n;
    dcc4.LineWidth  = n;
    dnc1.LineWidth  = n; dnc2.LineWidth  = n;dnc3.LineWidth  = n;
    dnc4.LineWidth  = n;
```

En las siguientes líneas de código se asigna las etiquetas correspondientes, y se las pone fuera de la gráfica debido al tamaño que ocupa, finalmente se le asigna un título a la gráfica.

```
    legend([dcc1,dcc2,dcc3,dcc4,dnc1,dnc2,dnc3,dnc4],{'clase 1','clase 2',...
   'clase 3','clase 4','Datos erróneos de clase 1', 'Datos erróneos de clase
2',...
'Datos erróneos de clase 3','Datos erróneos de clase 4'},'Location',
'eastoutside')
    title('Muestras de red neuronal con 4 clases')
```

En las siguientes líneas de código se aplica la misma receta para un dataset de dos clases.

```
else
    % muestras predichas correctamente de clase 1
    dcc1 = scatter(XY_Test(1,eti_Train==0 & eti_Train==eti_Test),...
           XY_Test(2,eti_Train==0 & eti_Train==eti_Test),...
           ele,eti_Test(eti_Train==0 & eti_Train==eti_Test));
    % muestras predichas correctamente de clase 2
    dcc2 = scatter(XY_Test(1,eti_Train==1 & eti_Train==eti_Test),...
           XY_Test(2,eti_Train==1 & eti_Train==eti_Test),...
           ele,eti_Test(eti_Train==1 & eti_Train==eti_Test));
    % muestras predichas incorrectamente de clase 1
    dnc1 = scatter(XY_Test(1,eti_Test==0 & eti_Train~=eti_Test),...
           XY_Test(2,eti_Test==0 & eti_Train~=eti_Test),...
           ele,eti_Test(eti_Test==0 & eti_Train~=eti_Test),'*');
    % muestras predichas incorrectamente de clase 2
    dnc2 = scatter(XY_Test(1,eti_Test==1 & eti_Train~=eti_Test),...
           XY_Test(2,eti_Test==1 & eti_Train~=eti_Test),...
           ele,eti_Test(eti_Test==1 & eti_Train~=eti_Test),'*');

    dcc1.LineWidth  = n;
    dcc2.LineWidth  = n;
    dnc1.LineWidth  = n;
    dnc2.LineWidth  = n;
 legend([dcc1,dcc2,dnc1,dnc2],{'clase 1','clase 2','Datos erróneos de clase
1',...
        'Datos erróneos de clase 2'},'Location','eastoutside')
    title('Muestras de red neuronal con 2 clases')
end
hold off
```

Resultado cuando se tienen 2 clases se puede observar en la Figura 1.24.

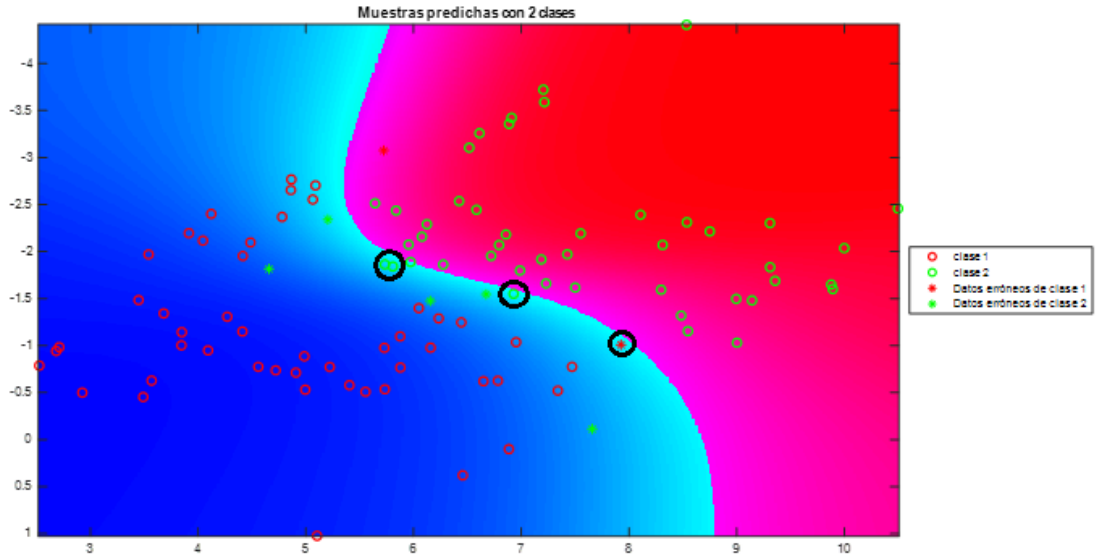


Figura 2.22 Ejemplo de Función Graficar

2.3.2. FUNCIÓN "ALEJAMIENTO"

Esta función sirve para tener una representación gráfica de las salidas de la red neuronal y de algoritmos en general cuando estas aún no han sido etiquetadas. La función no tiene salidas debido a que solo grafica las muestra, pero tiene tres entradas: "yreal", "ypredecir" y "ysinetiquetar".

yreal: corresponde a las etiquetas reales de los datasets.

ypredecir: corresponde a las etiquetas predichas por el algoritmo.

ysinetiquetar: corresponde a la salida del algoritmo, en el caso de la MLP, la probabilidad de que pertenezca a la clase

Como se pueden ver en las líneas de código, se guarda en la variable "m_correcta" el número de aciertos del algoritmo.

```
function alejamiento(yreal, ypredecir, ysinetiquetar)
% yreal:  salida real
% ypredecir: salida predicha por la red, pero etiquetada
% ysinetiquetar: salida de la red
mbien = length (ysinetiquetar(yreal == ypredecir,:));
```

Como se pueden ver en las líneas de código se guarda el valor máximo de cada clase y en la variable "media_Salida" se guarda el valor medio del vector que se forma de la variable anterior. Posteriormente se hace un vector que servirá del eje x en las gráficas.

```
vasalida = max(ysinetiquetar, [],2);
media_Salida = mean(vasalida)
xtotal = 1:1: length(m_correcta);
```

Como se pueden ver en las líneas de código se grafica los valores de "`mbien`", posteriormente se pone el título las leyendas y finalmente un texto sobre la cantidad de aciertos en la prueba correspondiente.

```
figure
hold on
plot(xtotal, salida, 'o')
title ('Valores de salida de todas las muestras')
legend (['Media de Salidas: ' num2str(media_Sal)],'Location','eastoutside')
annotation('textbox', 'String',['Aciertos: ' num2str(m_ mbien)])
hold off
end
```

Resultado cuando se tienen 2 clases se puede observar en la Figura 2.23:

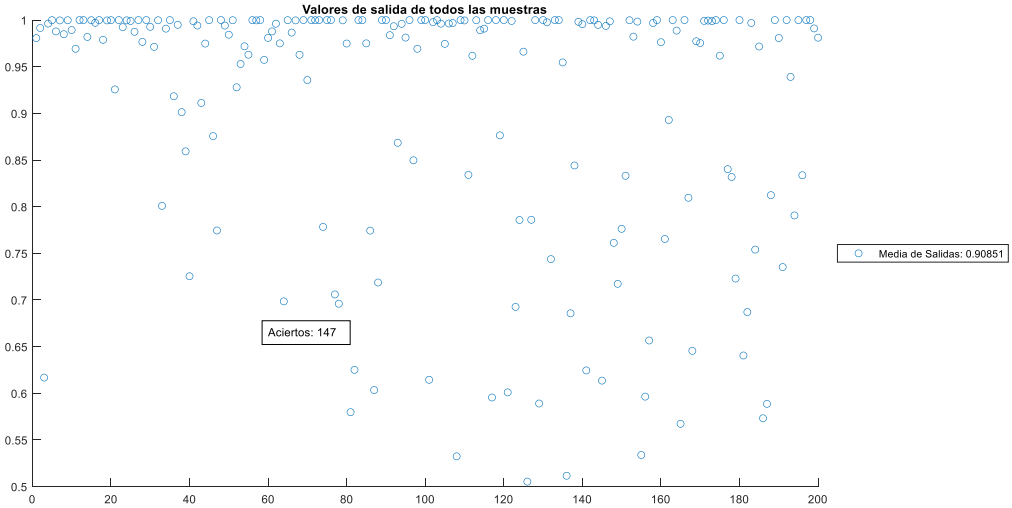


Figura 2.23 Que tan alejadas estan de la frontera de de decisión las muestras

2.3.3. FUNCIÓN CONFUSIÓN

En realidad, no es una función, es un método de una serie de clases. Que ha sido creado por Er.Abbas Manthiri S. del cual se toma como referencia para tomar las métricas que nos interesa.

Las líneas que se agrega a este código son:

```
disp('Métricas:')
                fprintf ('Exactitud: %f \n',Result.Accuracy)
                fprintf ('Precisión: %f \n',Result.Precision)
                fprintf ('Sensibilidad: %f \n',Result.Sensitivity)
                fprintf ('Especificidad: %f \n',Result.Specificity)
                fprintf ('Medida F1: %f \n',Result.F1_score)
```

Estas se encuentran en las líneas 129 a la 135 de la clase “confusión”.

- Función “Confusión” que es obtenida de una contribución de la página oficial de Matlab, más información se puede encontrar en [40].
- Función “alejamiento”

2.3.4. MULTI-LAYER PERCEPTRON

A continuación, se explican las líneas de código de la implementación de la MLP.

2.3.4.1. Código principal “MLPmain”

El código se basa en la sección 1.3.2.1. En la Figura 2.24 se puede observar que se utilizará una MLP con una capa oculta y en ella 16 neuronas seguida por una capa de salida que consta de 2 neuronas y se conectan directamente con las salidas las cuales contienen la probabilidad de que el elemento que entra en la red pertenezca a una u otra clase.

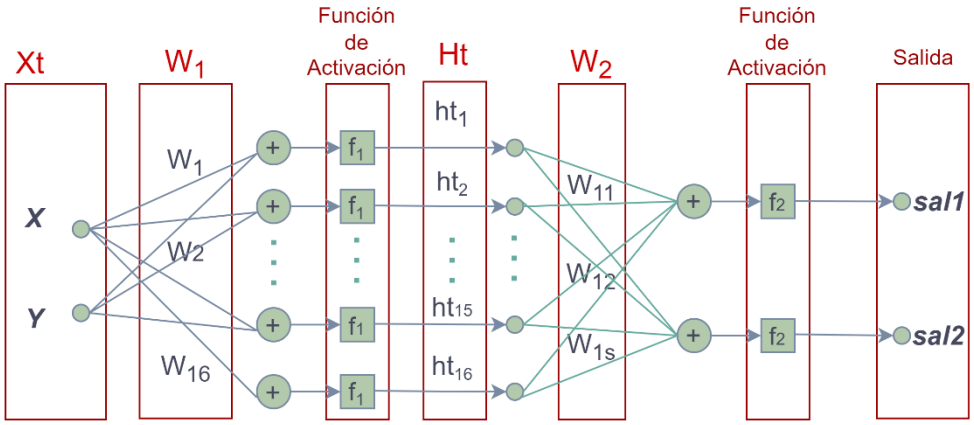


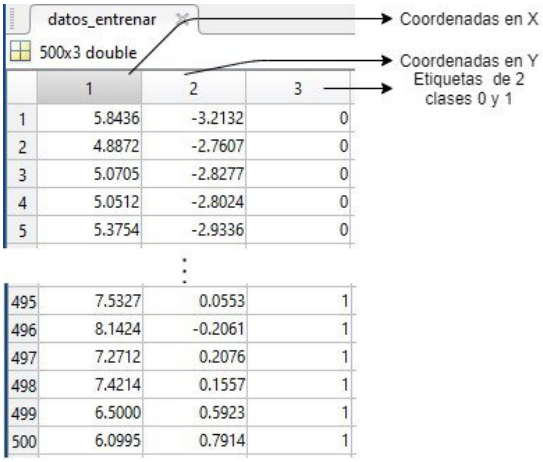
Figura 2.24 Modelo MLP empleado en el código.

Se carga los datos de entrenamiento que están en el archivo “Entrenamiento.mat”, en la sección 2.1.3 se explica los datasets que se pueden encontrar en este.

```
load('Entrenamiento.mat');
```

Se asigna a un conjunto de datos de entrenamiento a una matriz (Figura 2.25).

```
datos_entrenar = train2c200;
```

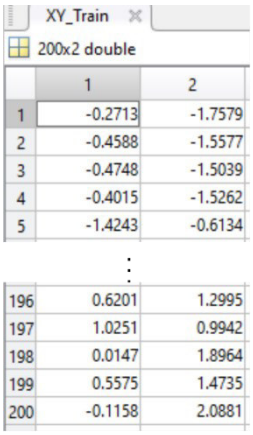


	1	2	3
1	5.8436	-3.2132	0
2	4.8872	-2.7607	0
3	5.0705	-2.8277	0
4	5.0512	-2.8024	0
5	5.3754	-2.9336	0
⋮			
495	7.5327	0.0553	1
496	8.1424	-0.2061	1
497	7.2712	0.2076	1
498	7.4214	0.1557	1
499	6.5000	0.5923	1
500	6.0995	0.7914	1

Figura 2.25 Ingreso de datos de entrenamiento

Se separa las variables de entrada y las se normaliza con la función z-score para que la relación en los datos tenga una media de cero y una varianza de 1 (Figura 2.26).

```
XY_Train = zscore(datos_entrenar(:,1:2));
```

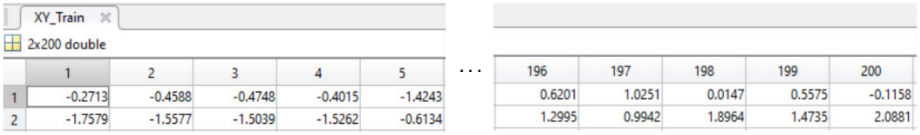


	1	2
1	-0.2713	-1.7579
2	-0.4588	-1.5577
3	-0.4748	-1.5039
4	-0.4015	-1.5262
5	-1.4243	-0.6134
⋮		
196	0.6201	1.2995
197	1.0251	0.9942
198	0.0147	1.8964
199	0.5575	1.4735
200	-0.1158	2.0881

Figura 2.26 Coordenadas de en “x” y “y” de cada punto

Posteriormente se ordenan los datos para realizar operaciones en la red neuronal (Figura 2.26).

```
XY_Train = XY_Train.';
```

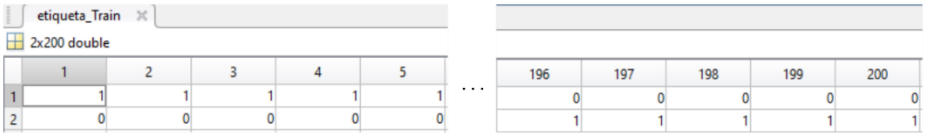


	1	2	3	4	5	⋯	196	197	198	199	200
1	-0.2713	-0.4588	-0.4748	-0.4015	-1.4243		0.6201	1.0251	0.0147	0.5575	-0.1158
2	-1.7579	-1.5577	-1.5039	-1.5262	-0.6134		1.2995	0.9942	1.8964	1.4735	2.0881

Figura 2.27 Matriz de entrada transpuesta

Luego se separa las salidas deseadas y se convierte en una matriz para tener un número de salidas en función del número de clases (Figura 2.27).

```
etiqueta_Train = Amatriz(datos_entrenar(:,3));
etiqueta_Train = etiqueta_Train.';
```

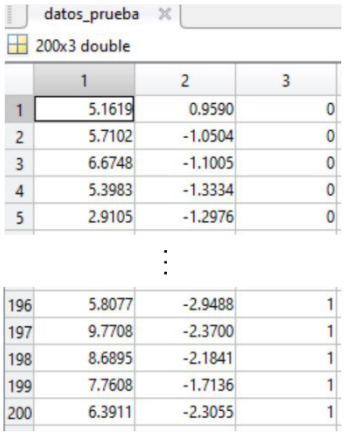


	1	2	3	4	5	⋯	196	197	198	199	200
1	1	1	1	1	1		0	0	0	0	0
2	0	0	0	0	0		1	1	1	1	1

Figura 2.28 Etiquetas de datos de entrenamiento

Para los Datos de prueba se sigue el mismo proceso que obtuvimos para el dataset de entrenamiento como se observa en la Figura 2.29, Figura 2.30, Figura 2.31 y Figura 2.32.

```
%% Cargar Dataset de Prueba
load('Prueba.mat');
datos_prueba = test2c;
```



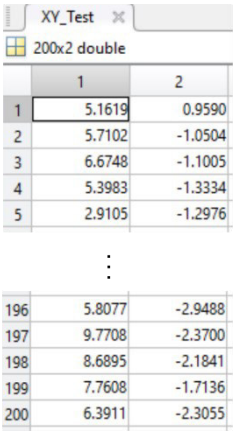
datos_prueba

200x3 double

	1	2	3
1	5.1619	0.9590	0
2	5.7102	-1.0504	0
3	6.6748	-1.1005	0
4	5.3983	-1.3334	0
5	2.9105	-1.2976	0
⋮			
196	5.8077	-2.9488	1
197	9.7708	-2.3700	1
198	8.6895	-2.1841	1
199	7.7608	-1.7136	1
200	6.3911	-2.3055	1

Figura 2.29 Dataset de prueba

```
XY_Test = datos_prueba(:,1:2);
```



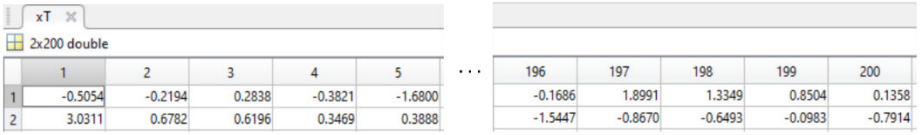
XY_Test

200x2 double

	1	2
1	5.1619	0.9590
2	5.7102	-1.0504
3	6.6748	-1.1005
4	5.3983	-1.3334
5	2.9105	-1.2976
⋮		
196	5.8077	-2.9488
197	9.7708	-2.3700
198	8.6895	-2.1841
199	7.7608	-1.7136
200	6.3911	-2.3055

Figura 2.30 Coordenadas de cada muestra del dataset de prueba

```
xT = zscore(XY_Test);
xT = xT.';
```



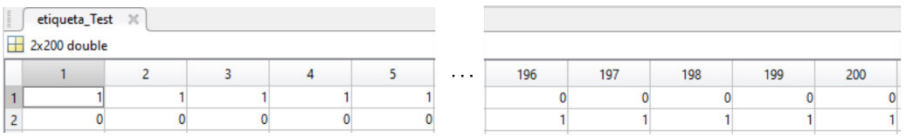
xT

2x200 double

	1	2	3	4	5	⋯	196	197	198	199	200
1	-0.5054	-0.2194	0.2838	-0.3821	-1.6800		-0.1686	1.8991	1.3349	0.8504	0.1358
2	3.0311	0.6782	0.6196	0.3469	0.3888		-1.5447	-0.8670	-0.6493	-0.0983	-0.7914

Figura 2.31 Matriz de entrada transpuesta

```
etiqueta_Test = Amatriz(datos_prueba(:,3));
etiqueta_Test = etiqueta_Test.';
```



etiqueta_Test

2x200 double

	1	2	3	4	5	⋯	196	197	198	199	200
1	1	1	1	1	1		0	0	0	0	0
2	0	0	0	0	0		1	1	1	1	1

Figura 2.32 Etiquetas de cada muestra

A continuación, como se observa en la Figura 2.33 se extrae las dimensiones de la matriz que contiene las muestras, seguido se extraen las dimensiones de la matriz que contiene las etiquetas como se aprecia en la Figura 2.34.

```
inputs = size(xTRain,1);
```

```
inputs =

     2
```

Figura 2.33 Número de dimensiones con las que se trabajará

```
[outputs,patterns] = size(yTrain);
```

```
outputs =        patterns =

     2                200
```

Figura 2.34 Número de dimensiones y número de muestras de entrenamiento

En “hidden” se asigna el número de neuronas de la capa oculta en base a la Figura 2.24.

```
hidden = 16;
```

En la Figura 2.35 se puede ver cómo se crean las matrices de pesos y valores para el bias, empezando con valores aleatorios, entre -1 y 1. En la Figura 2.36 se observa la matriz de pesos de la capa de salida y en la Figura 2.37 se observar la matriz que contiene el bias.

```
% Creación de pesos y bias
W1 = rand(hidden,inputs+1)*2-1; % Capas ocultas
```

W1 — 16x3 double

	1	2	3
1	-0.2838	0.2804	0.0351
2	0.9739	-0.6741	0.2540
3	-0.8320	0.1318	0.8264
4	-0.4994	0.8632	0.3279
5	0.6227	0.5662	-0.2216
6	-0.8312	0.3714	0.4800
7	0.0625	-0.0676	0.6353
8	0.6012	-0.4794	0.2007
9	0.4776	0.1385	-0.8300
10	-0.7167	-0.5025	0.8447
11	-0.1242	-0.3614	-0.8928
12	-0.2992	0.8216	0.0540
13	-0.0430	0.7704	-0.7623
14	0.1748	0.5892	-0.2397
15	-0.7084	0.8516	0.6257
16	0.8107	-0.6423	-0.5118

Figura 2.35 Matriz de pesos de capa oculta

```
W2 = rand(outputs,hidden+1)*2-1; % última capa
```

W2 — 2x17 double

	1	2	3	4	5	…	13	14	15	16	17
1	0.7688	-0.2437	-0.4943	-0.9003	0.2406		0.5813	0.3562	0.6023	0.8920	0.8169
2	0.4253	-0.5022	0.5345	0.3706	0.4934		-0.5910	-0.8950	0.3571	-0.8169	0.0199

Figura 2.36 Matriz de pesos de capa de salida

```
X=[ones(1,patterns)*2-1; xTRain];     %Entrada con bias
```

X — 3x200 double

	1	2	3	4	5	…	196	197	198	199	200
1	2	2	2	2	2		2	2	2	2	2
2	-0.2713	-0.4588	-0.4748	-0.4015	-1.4243		0.6201	1.0251	0.0147	0.5575	-0.1158
3	-1.7579	-1.5577	-1.5039	-1.5262	-0.6134		1.2995	0.9942	1.8964	1.4735	2.0881

Figura 2.37 Bías

Se elige el número máximo de ciclos con el que se entrenará la red, con un learning rate fijo para todos los casos y posteriormente se inicializa el bias.

```
maxcycles=3000; %número máximo de ciclos de entrenamiento
lr = 0.0001;        %Tasa de aprendizaje
```

A continuación, se empieza a entrenar la red con el número de ciclos implementado previamente.

```
%Algoritmo de Back-propagation
tic
for i=1:maxcycles
```

Como se observa en la Figura 2.38, e calcula las salidas de la capa oculta, en la variable "h" se guarda una matriz con las salidas de la capa oculta y agregando un vector de 1s debido a que el bias ya está integrado en la salida h de la capa oculta. Posteriormente se calcula el error de la capa de salida (Figura 2.39) y posteriormente su respectivo MSE. (Figura 2.40)

```
h = sigmoid(W1*X);
```

h

16x200 double

	1	2	3	4	5	⋯	196	197	198	199	200
1	0.3307	0.3206	0.3201	0.3244	0.2712		0.4139	0.4390	0.3783	0.4111	0.3713
2	0.8434	0.8654	0.8683	0.8618	0.9400		0.8653	0.8189	0.9183	0.8750	0.9280
3	0.0410	0.0469	0.0488	0.0484	0.0864		0.3756	0.3302	0.4763	0.4078	0.5116
4	0.1407	0.1295	0.1299	0.1364	0.0810		0.4907	0.5528	0.4100	0.4914	0.3980
5	0.8148	0.7910	0.7875	0.7952	0.6399		0.7873	0.8328	0.6971	0.7746	0.6720
⋮						⋱			⋮		
12	0.2857	0.2574	0.2554	0.2668	0.1416		0.4953	0.5738	0.3813	0.4848	0.3588
13	0.7398	0.6787	0.6670	0.6831	0.3283		0.3546	0.4865	0.1794	0.3144	0.1459
14	0.6482	0.6113	0.6060	0.6175	0.4152		0.5995	0.6716	0.4759	0.5805	0.4454
15	0.0602	0.0583	0.0594	0.0622	0.0468		0.4811	0.5195	0.4458	0.4950	0.4480
16	0.9367	0.9378	0.9368	0.9346	0.9453		0.6360	0.6116	0.6550	0.6246	0.6518

Figura 2.38 Salida de primera capa

```
H=[ones(1,patterns);h];

e=yTrain-sigmoid(W2*H);
```

e

2x200 double

	1	2	3	4	5	⋯	196	197	198	199	200
1	0.1191	0.1308	0.1325	0.1287	0.2042		-0.8615	-0.8902	-0.8015	-0.8528	-0.7840
2	-0.5402	-0.5584	-0.5612	-0.5558	-0.6425		0.4289	0.4699	0.3799	0.4199	0.3715

Figura 2.39 Error en la salida de la red neuronal

```
MSE(i)= mean(mean(e.^2));
```

MSE

1x3000 double

	1	2	3	4	5	⋯	2996	2997	2998	2999	3000
1	0.3649	0	0	0	0		0	0	0	0	0

Figura 2.40 MSE de la red neuronal

Si se está en la primera iteración, se actualiza los pesos normalmente, caso contrario se ejecuta el algoritmo para actualizar el learning rate.

```
if i == 1
```

En la Figura 2.41 se puede observar la salida de la red neuronal sin etiquetar.

```
eti_entre = sigmoid(W2*H);
```

eti_entre

2x200 double

	1	2	3	4	5	...	196	197	198	199	200
1	0.8809	0.8692	0.8675	0.8713	0.7958		0.8615	0.8902	0.8015	0.8528	0.7840
2	0.5402	0.5584	0.5612	0.5558	0.6425		0.5711	0.5301	0.6201	0.5801	0.6285

Figura 2.41 Salida de la red neuronal sin etiquetar

Se actualizan los pesos de la última capa como se logra observar en la Figura 2.42.

```
Actualización de pesos
delta2 = eti_entre.*(1-eti_entre).*e;
 del_W2 = 2*lr* delta2*H';
 W2 = W2+ del_W2; %actualización de última capa
```

W2

2x17 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0.7583	-0.2482	-0.5021	-0.9012	0.2363	0.9448	-0.4813	0.6085	-0.8247	-0.6347	-0.2899	0.4475	0.5755	0.3474	0.5940	0.8891	0.8086
2	0.4202	-0.5020	0.5265	0.3671	0.4958	-0.2317	0.7540	-0.0851	0.1207	0.0651	-0.3778	0.0319	-0.5872	-0.8875	0.3603	-0.8162	0.0146

Figura 2.42 Actualización de pesos de última capa

Se actualizan los pesos de la última capa como se logra observar en la Figura 2.43.

```
delta1 = h.*(1-h).*(W2(:,2:hidden+1)'*delta2);
del_W1 = 2*lr*delta1*X';
 W1 = W1 + del_W1; %actualización de capas ocultas
```

W1

16x3 double

	1	2	3
1	-0.2818	0.2772	0.0377
2	0.9777	-0.6709	0.2515
3	-0.8320	0.1351	0.8240
4	-0.5003	0.8655	0.3256
5	0.6221	0.5632	-0.2188
6	-0.8310	0.3758	0.4764
7	0.0596	-0.0699	0.6372
8	0.6057	-0.4768	0.1984
9	0.4789	0.1403	-0.8312
10	-0.7136	-0.5039	0.8460
11	-0.1263	-0.3623	-0.8919
12	-0.3024	0.8167	0.0589
13	-0.0430	0.7653	-0.7581
14	0.1708	0.5903	-0.2404
15	-0.7116	0.8453	0.6307
16	0.8080	-0.6439	-0.5109

Figura 2.43 Actualización de capa oculta

```
   continue
else
```

Si el MSE actual es mayor o igual al MSE anterior entonces se disminuye el learning rate, caso contrario se aumenta el learning rate y se actualiza los pesos.

```
% Learning rate adaptativo
 if MSE(i) >= MSE(i-1)
    lr = lr*0.9;
 elseif MSE(i) < MSE(i-1)
   lr = lr*1.0001;
   % Actualización de pesos
```

En “eti_entre” se guardan las salidas de la red neuronal.

```
eti_entre = sigmoid(W2*H);
delta2 = eti_entre.*(1-eti_entre).*e;
del_W2 = 2*lr* delta2*H';
```

Se actualizan los pesos a medida que pasa cada iteración, en la Figura 2.44 se puede ver como quedan los pesos de la capa de salida después del entrenamiento.

W1 W2

2x17 double

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	-0.5573	0.9201	-3.9366	2.4994	1.6373	-2.2349	-1.1518	-1.5726	-1.6658	3.2504	1.9517	3.2499	-0.5708	-0.8500	0.9243	-4.2670	2.7437
2	0.8892	-0.4452	4.0739	-2.3784	-1.9703	1.1377	0.1785	1.3642	1.6736	-2.7887	-2.3192	-2.4115	0.3084	0.6437	-2.0007	4.8531	-1.9988

Figura 2.44. Pesos de la capa de salida actualizados

```
W2 = W2+ del_W2; %actualización de última capa
delta1 = h.*(1-h).*(W2(:,2:hidden+1)'*delta2);
del_W1 = 2*lr*delta1*X';
```

Se actualizan los pesos a medida que pasa cada iteración, en la Figura 2.45 se puede ver como quedan los pesos de la capa oculta después del entrenamiento.

```
W1 = W1+del_W1; %actualización de capa oculta
```

W1 W2

16x3 double

	1	2	3
1	0.2925	-0.8103	-0.3007
2	-0.2080	-2.4312	-6.0370
3	0.0371	0.9646	4.9117
4	-1.4944	-2.0314	-1.5733
5	-1.1959	1.8418	1.2502
6	0.0857	0.4576	0.7516
7	0.9518	1.4310	0.9690
8	0.0194	0.2195	-3.6788
9	-1.8121	-2.8249	-1.6644
10	0.6313	-1.7470	-0.3361
11	0.1459	2.2170	4.8816
12	-1.3617	0.0817	-0.0601
13	-0.1833	0.5256	0.9059
14	-0.8675	-1.4564	-0.8129
15	-2.7571	4.4704	2.8494
16	-0.6559	-1.6470	-0.4027

Figura 2.45 Pesos de la capa oculta actualizados

```
        end
      end

end;clf
```

En las siguientes se extrae los resultados de la red con la variable `etiqueta_Train`, donde se extraen los máximos valores de cada fila de una matriz que se forma con los valores de salida para cada muestra, para determinar a qué clase pertenece (Figura 2.46).

```
[~, eti_entre]= max(eti_entre.',[],2);
[~, etiqueta_Train]= max(etiqueta_Train.',[],2);
```

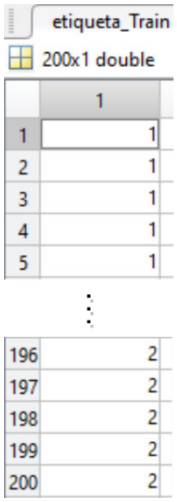


Figura 2.46 A la izquierda salida de la red neuronal y a la derecha etiqueta original

Se vuelve a sacar el tamaño de los datos de entrada y salida para poder operar en la red neuronal.

```
% Evaluación de Red Neuronal con muestras de prueba.
inputs = size(xT,1);
[outputs,patterns] = size(xT);
```

Se calcula la salida con los datos de prueba y con los valores de los pesos de la red entrenada de la misma manera que en el dataset de entrenamiento.

```
Xt = [ones(1,patterns); xT];
ht = sigmoid(W1*Xt);
Ht = [ones(1,patterns);ht];
Ypredecir = sigmoid(W2*Ht);
Ypredecir = Ypredecir.';
etiqueta_Test = etiqueta_Test.';
```

Se extrae las salidas de los resultados obtenidos y de los resultados esperados.

```
[~, eti_prueba] = max(Ypredecir,[],2);
[~, etiqueta_Test] = max(etiqueta_Test,[],2);
```

Posteriormente se usa la función de Matlab "`confusionchart`" en ella se ingresa los valores predichos y los valores reales. Los demás términos de entrada de la red son para ampliar la información de la matriz de confusión.(Figura 2.47)

```
%% Resultados
% Matriz de confusión de datos de prueba
figure
cm = confusionchart(etiqueta_Test,eti_prueba,'RowSummary','row-normalized',...
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
```

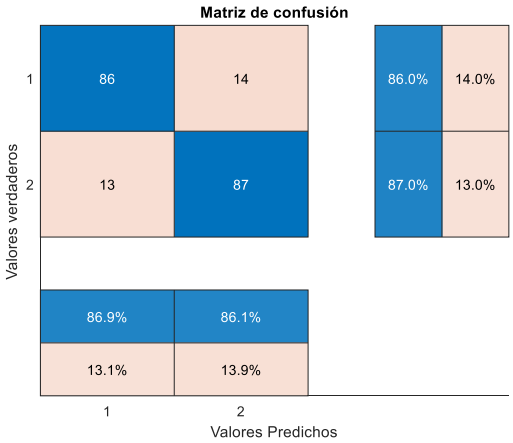


Figura 2.47 Matriz de confusión evaluando el dataset de prueba

Se calculan los resultados de las métricas para obtener más información del algoritmo en el dataset como se observa en la Figura 2.48.

```
% Resultados de muestras de entrenamiento
[~,Result]= confusion.getMatrix(etiqueta_Train,eti_entre);
fprintf('Porcentaje de aciertos para dataset de entrenamiento es: %.2f%% \n ',...
    Result.Accuracy*100);
```

```
Métricas:
Exactitud: 0.850000
Precisión: 0.857143
Sensibilidad: 0.840000
Especificidad: 0.860000
Medida F1: 0.848485
Porcentaje de aciertos para dataset de entrenamiento es: 85.00%
```

Figura 2.48 Resultados de dataset de entrenamiento

En las siguientes líneas de código se puede ver cómo se obtienen las métricas necesarias para evaluar el modelo de clasificación, los términos de entrada son "`etiqueta_Test`" y "`eti_prueba`" que representan los valores reales y los valores predichos respectivamente. El resultado se puede apreciar en la Figura 2.49.

```
% Resultados de muestras de prueba
[c_matrix,Result,RefereceResult] = confusion.getMatrix(etiqueta_Test,eti_prueba);
fprintf('Porcentaje de aciertos para dataset prueba es: %.2f%% \n',...
    Result.Accuracy*100);
```

```
Métricas:
Exactitud: 0.865000
Precisión: 0.868687
Sensibilidad: 0.860000
Especificidad: 0.870000
Medida F1: 0.864322
Porcentaje de aciertos para dataset prueba es: 86.50%
```

Figura 2.49 Resultados de dataset de prueba

Se crea una malla para que se vea de una manera intuitiva, los datos clasificados a la salida de la red.

```
% Creación de malla para observar la frontera de decisión del clasificador
```

Primero se escogen los mínimos y los máximos de las muestras, posteriormente se hacen vectores de 500 elementos entre los mínimos valores en los distintos ejes. Posteriormente con estos valores se crean 2 vectores uno para el eje “x” y el otro para el eje “y”, cada uno con 100 muestras. Estos se pueden apreciar en la Figura 2.50 y en la Figura 2.51.

```
minimos = min(XY_Test);
maximos = max(XY_Test);
```

```
minimos =

   -7.9800  -43.0637

maximos =

    8.0000   47.7903
```

```
mallax = linspace(minimos(1),maximos(1),100);
```

mallax — 1x100 double

	1	2	3	4	5	…	95	96	97	98	99	100
1	2.1622	2.2485	2.3348	2.4211	2.5074	…	10.2749	10.3612	10.4475	10.5338	10.6201	10.7064

Figura 2.50 Datos en x de la malla que se creara para evaluar el algoritmo en el eje x

```
mallay = linspace(minimos(2),maximos(2),100);
```

mallay — 1x100 double

	1	2	3	4	5	…	95	96	97	98	99	100
1	-4.1050	-4.0538	-4.0027	-3.9515	-3.9004	…	0.7032	0.7544	0.8055	0.8567	0.9078	0.9590

Figura 2.51 Datos en x de la malla que se creara para evaluar el algoritmo en el eje y

Se agrupan en una variable llamada coordenadas la cual ingresaremos para dibujar cada punto. (Figura 2.52)

```
coordenadas = [mallax.' mallay.'];
```

coordenadas — 100x2 double

	1	2
1	2.1622	-4.1050
2	2.2485	-4.0538
3	2.3348	-4.0027
4	2.4211	-3.9515
5	2.5074	-3.9004
⋮		
95	10.2749	0.7032
96	10.3612	0.7544
97	10.4475	0.8055
98	10.5338	0.8567
99	10.6201	0.9078
100	10.7064	0.9590

Figura 2.52 Coordenadas de cada punto de la malla

Se crea la malla con la función “meshgrid”, la cual devuelve coordenadas 2-D basadas en las coordenadas contenidas en los vectores “x” y “y”. Se puede ver del eje “x” en la Figura 2.53 y del eje “y” en Figura 2.54. La malla completa se puede apreciar en la Figura 2.55.

```
[u, v] = meshgrid(mallax, mallay);
```

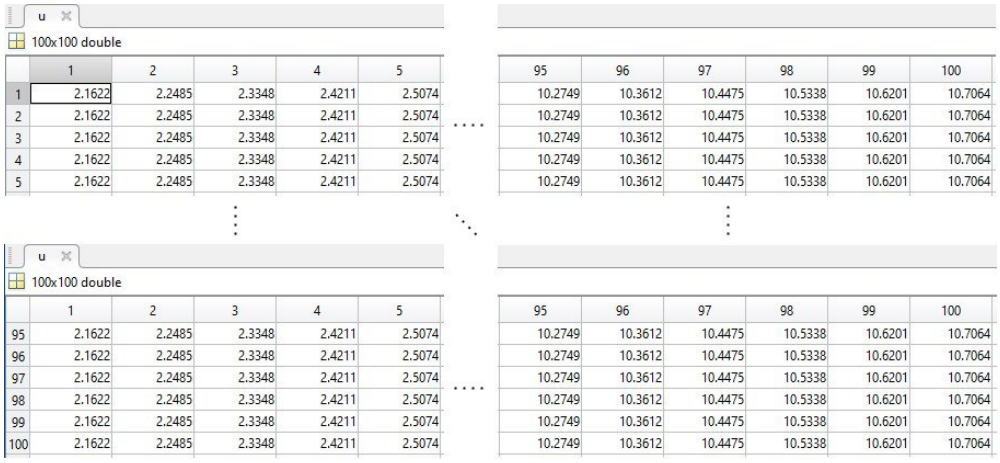


Figura 2.53 Valor de cada punto en eje x

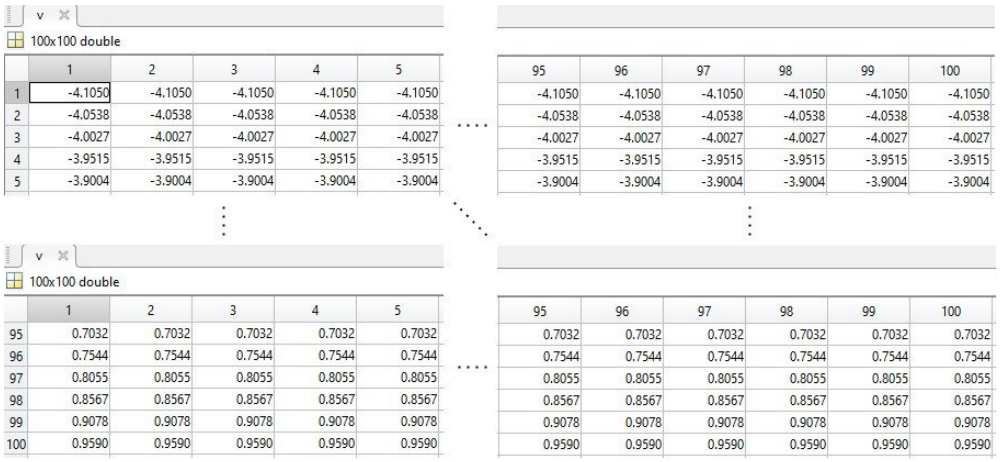


Figura 2.54 valor de cada punto en eje y

```
malla = [u(:)'; v(:)'];
```

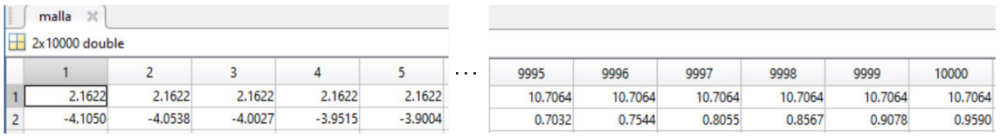


Figura 2.55 Malla adecuada para ingresar en el algoritmo a probar.

Se normalizan los datos con la función z-score y se ordena la matriz. (Figura 2.56)

```
mallaz = zscore(malla.');
mallaz = mallaz.';
```

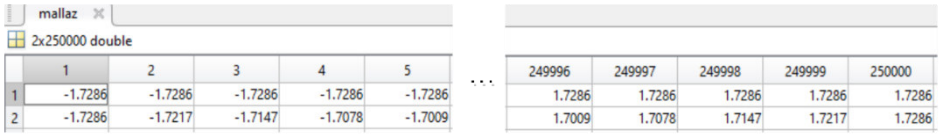


Figura 2.56 Malla con los cálculos para ingresar en red neuronal

Se prueba la MLP con la malla el resultado se encuentra en la variable “sal_malla”, y esa una salida que no está etiquetada todavía como se observa en la Figura 2.57.

```
% Aplicación de red neuronal en la malla
inputsm = size(mallaz,1);
[outputsm,patternsm] = size(mallaz);
Xt = [ones(1,patternsm); mallaz];
```

```
ht = sigmoid(W1*Xt);
Ht = [ones(1,patternsm);ht];
Ympro = sigmoid(W2*Ht);
sal_malla = Ympro.';
```

sal_malla

10000x2 double

	1	2
1	0.7357	0.2610
2	0.7373	0.2571
3	0.7390	0.2532
4	0.7406	0.2493
5	0.7422	0.2455
⋮		
9996	0.1873	0.7433
9997	0.1882	0.7402
9998	0.1891	0.7370
9999	0.1900	0.7337
10000	0.1909	0.7304

Figura 2.57 Salida de malla para darle al gráfico el efecto de degradado.

En “Ympro” se guardará losa valores máximos de las salidas, es por ellos que se elige la primera salida de la función “max”, y en “Ymalla” se tiene las salidas categóricas de la red, estas dos son sumadas y se da una nueva forma a la matriz para que pueda ser insertada normalmente en la función graficar. Para dar matices al gráfico se suma “sal_malla” más “Ypro”, para que en el gráfico se dé un efecto de degradado. Como se puede ver en la Figura 2.58.

```
[Ympro, sal_malla] = max(sal_malla,[],2);
sal_malla = sal_malla + Ympro;
sal_malla = reshape(sal_malla, size(u,1),size(u,2));
```

sal_malla

100x100 double

	1	2	3	4	5	⋯	96	97	98	99	100
1	1.7357	1.7277	1.7193	1.7106	1.7014		2.9908	2.9911	2.9914	2.9917	2.9919
2	1.7373	1.7294	1.7210	1.7123	1.7032	⋯	2.9908	2.9911	2.9914	2.9917	2.9920
3	1.7390	1.7310	1.7227	1.7141	1.7050		2.9908	2.9911	2.9914	2.9917	2.9920
4	1.7406	1.7327	1.7244	1.7158	1.7068		2.9908	2.9911	2.9914	2.9917	2.9920
5	1.7422	1.7343	1.7261	1.7176	1.7086		2.9908	2.9911	2.9914	2.9917	2.9920
⋮					⋱				⋮		
96	1.9917	1.9913	1.9909	1.9905	1.9901		2.7111	2.7196	2.7278	2.7357	2.7433
97	1.9916	1.9912	1.9908	1.9904	1.9900		2.7076	2.7162	2.7245	2.7325	2.7402
98	1.9915	1.9911	1.9907	1.9903	1.9899	⋯	2.7042	2.7128	2.7212	2.7292	2.7370
99	1.9914	1.9910	1.9906	1.9902	1.9897		2.7007	2.7094	2.7178	2.7259	2.7337
100	1.9913	1.9909	1.9905	1.9901	1.9896		2.6971	2.7059	2.7144	2.7226	2.7304

Figura 2.58 salida del algoritmo en cada punto.

Se grafican las muestras etiquetadas de acuerdo con el algoritmo elegido como se puede ver en la Figura 2.59.

```
% Graficas de resultado de datos de prueba
Graficar(eti_prueba.'-1, etiqueta_Test.'-1, XY_Test.',sal_malla, coordenadas)
```

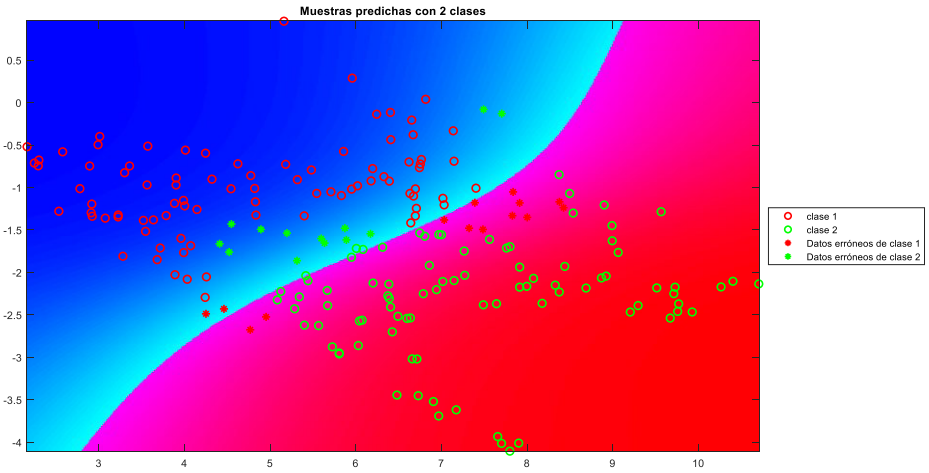


Figura 2.59 Gráfica ilustrativa de la clasificación que hace el algoritmo

Finalmente, se grafican las salidas de la red antes de etiquetar con la esperanza de encontrar algún patrón a cuando se prueban los datasets basándose en la lejanía de la frontera de decisión. (Figura 2.60)

```
alejamiento(etiqueta_Test, eti_prueba, Ypredecir)
```

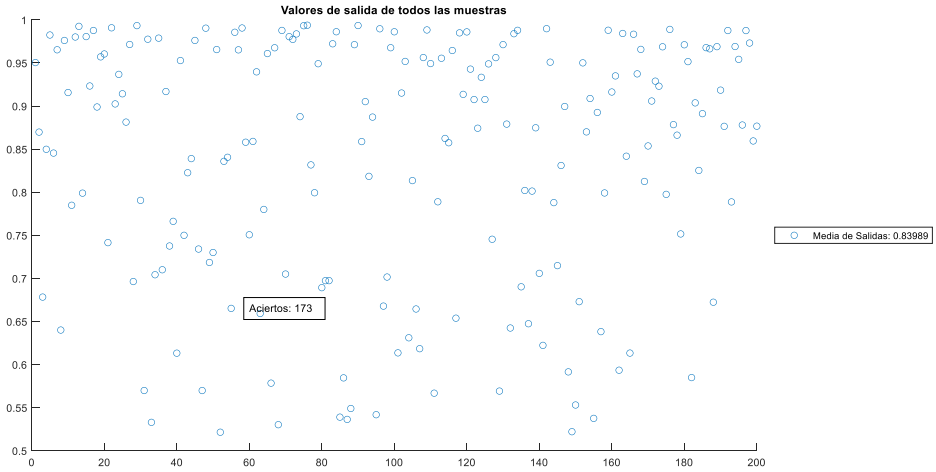


Figura 2.60 Gráfica que ilustra el grado de alejamiento de cada muestra de prueba de la frontera de decisión, según el algoritmo.

2.3.5. SUPPORT VECTOR MACHINE

A continuación, se explica el código acerca del algoritmo SVM que se encuentra en la Sección Support Vector Machine (SVM)1.3.2.2. Como se puede apreciar en la Figura 1.13 lo que se pretende con este programa es encontrar los vectores de soporte en los cuales se va a basar el hiperplano que pretende encontrará el algoritmo.

- Función “Graficar” (ya ha sido explicada)
- Función “Confusión” (ya ha sido explicada)
- Función “alejamiento” (ya ha sido explicada)

- Función “mainSVM”, que es la función principal.
- Función “ajustarsvm”, es la función en donde se resuelve el problema de optimización cuadrática y se encuentran los lagrangianos que son importantes para encontrar los vectores de soporte.
- Función “Probarsvm”, sirve para probar el algoritmo previamente entrenado.
- Función “mainSVM”

2.3.5.1. Código principal “mainSVM”

Aquí se tiene el código principal, en las siguientes líneas se carga los datos en donde se separa las muestras de las clases y posteriormente se procede a usar la función “zscore” para normalizar las muestras en donde se tiene una media de cero y una varianza de uno. No se incluirán los gráficos ya que estos son similares a los del algoritmo MLP.

```
clc, clear, close all;
%% Ingreso de datos
load('Entrenamiento.mat');
entrenamiento = train2c200;
entrenamiento(:,1:end-1)=zscore(entrenamiento(:,1:end-1));
XY_Train = entrenamiento(:,1:end-1).';
etiqueta_Train = entrenamiento(:,end).';
```

Como ya se ha hecho, se carga el dataset de prueba, se normaliza, y se prueba con este modelo.

```
load('Prueba.mat');
prueba = test2c;
Xp = prueba(:,1:end-1);
prueba(:,1:end-1)=zscore(prueba(:,1:end-1));
XY_Test = prueba(:,1:end-1).';
etiqueta_Test = prueba(:,end);
```

Se inicializa la variable C que es la penalización por equivocación a la hora de entrenar el algoritmo, seguido se toma a “gamma” como 1 y es la constante usada en el kernel RBF, la variable grado corresponde a la constante usada para el kernel polinómico, finalmente se elige un kernel, pero se puede elegir entre `'g'`, que representa kernel gaussiano, `'p'`, que representa kernel polinomial y `'l'` que representa kernel lineal.

```
C  = 10 ;
gamma = 1;
grado = 3;
ker = 'g';
```

Se crea un vector donde se guardan las etiquetas de cada clase, como se observa en la Figura 2.61.

```
%% Formulas de kernel y entrenamiento
clases = unique(etiqueta_Train);
```

```
clases =

     0     1
```

Figura 2.61 Etiqueta de cada clase.

Se inicializa la estructura donde se guardará el modelo para cada una de las clases, además se inicializa un vector auxiliar donde se va a reasignar las etiquetas de los targets, debido a que se utilizará el algoritmo uno contra todos, es decir que la clase que le toque ser entrenada tendrá el valor de 1 y las demás -1.

```
if length(clases) > 2
    modelo{length(clases)} = [];
    yaux = zeros(size(etiqueta_Train));
```

El primero lazo es usado para recorrer el vector “clases”, debido a que este tiene los targets diferentes a ser utilizados, posteriormente se usa otro lazo que nos sirve para la reclasificación en el que la clase que se elige tendrá el valor de 1 y las demás clases -1. Se supone que le toca a la clase 2 entonces el valor que se asignará a esa clase es 1 mientras que las clases 1, 3 y 4 tendrán los valores de -1, y así para cada una de las clases.

```
    for i = 1: length(clases)
        for j = 1:length(etiqueta_Train)
            if etiqueta_Train(j) == clases(i)
                yaux(j) = 1;
            else
                yaux(j) = -1;
            end
        end
```

Se entrenará a cada clase con la función “ajustarsvm”, en donde:

A continuación, se observan los elementos de la función. En su salida se tiene al modelo ajustado en la estructura “`modelo`”, donde se guardarán todas las variables necesarias para probarlo. Se verá las entradas de la función:

X: representa los datos a clasificar (asteriscos de gráficas anteriores).

Kernel: el kernel con el cual se trabajará

y: representan a las salidas deseadas de los datos a clasificar.

C: representa la penalización cuando el algoritmo se equivoca en una muestra.

gamma: Es la constante entre cero y uno que se utilizan en el kernel de distribución gaussiana.

Grado: es la variable que se utiliza en el kernel polinómico.

En la Figura 2.62 se puede ver un ejemplo de la estructura “modelo”.

```
        modelo{i} = ajustarsvm(XY_Train,ker,yaux,C,gamma,grado);
    end
```

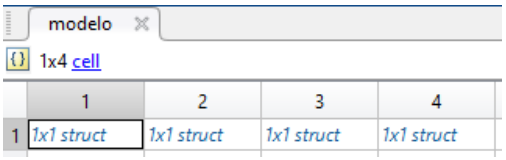


Figura 2.62 Modelo creado para guardar los valores de cada clasificador usado cuando se tiene más de 2 clases.

```
else
```

Si solo se tiene dos clases simplemente escala los targets a los valores de -1 y 1. (Figura 2.63)

```
yr = rescale(etiqueta_Train,-1,1);
```

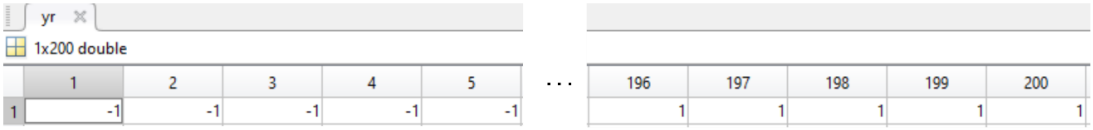


Figura 2.63 etiquetas transformadas a valores de -1 y 1

En la siguiente línea se llama a la función que crear el modelo basado en el algoritmo SVM. La estructura modelo se puede ver en la Figura 2.64 donde se puede apreciar los vectores y variables necesarias para hacer predicciones con nuevos datos.

```
modelo = ajustarsvm(XY_Train,ker,yr,C,gamma,grado);
```

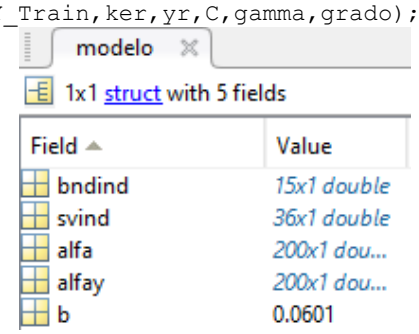


Figura 2.64 modelo donde se guardan los valores principales que construyó el algoritmo

```
end
```

Esta línea se usa para probar los datos de entrenamiento y de prueba.

En la primera gráfica de la Figura 2.65 , en “eti_entre” se tiene las etiquetas predichas con los mismos datos que fueron utilizados para ajustar el modelo. En la segunda gráfica con “eti_prueba” tenemos las salidas del algoritmo que ya han sido etiquetadas mientras que en la última gráfica se puede ver que en “ysinetiqueta” se tiene las salidas del modelo sin etiquetar.

```
% Evaluación muestras de entrenamiento
eti_entre = ProbarSVM(etiqueta_Train, XY_Train, XY_Train, modelo, ker,...
 gamma, grado);

% Evaluación muestras de prueba
[eti_prueba, ysinetiqueta] = ProbarSVM(etiqueta_Test, XY_Test, XY_Train,
 modelo, ker, gamma, grado);
```

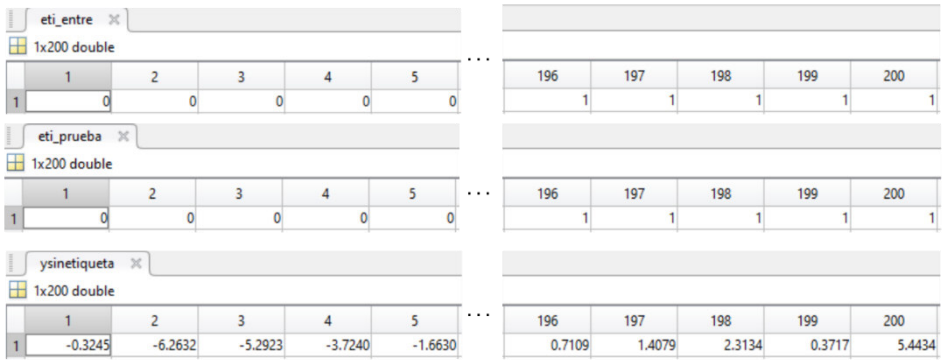


Figura 2.65 Salidas del modelo SVM

La matriz de confusión de la Figura 2.66 se realiza con el mismo código que se realizó la matriz de confusión para la MLP.

A continuación, se presenta el código para mostrar la matriz de confusión con su respectiva gráfica en la Figura 2.66.

```
%% Resultados
% Matriz de confusión de datos de prueba
figure
cm = confusionchart(etiqueta_Test,eti_prueba,'RowSummary','row-normalized',...
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
```

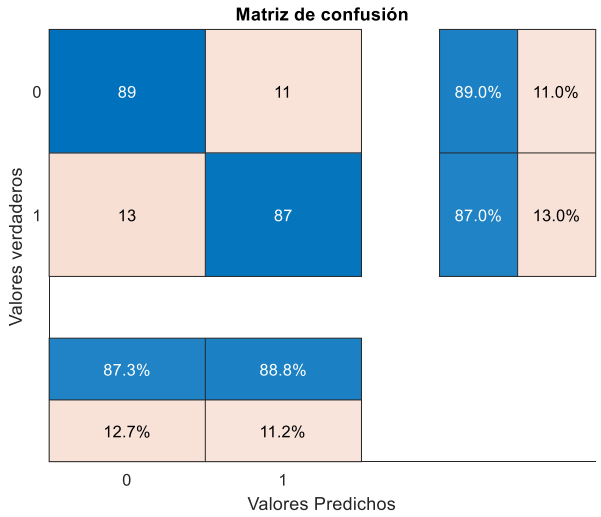


Figura 2.66 Matriz de confusión con SVM

Se calcula los resultados de las métricas para obtener más información del algoritmo en el dataset , al igual que se hizo con la MLP.

```
% Resultados de muestras de entrenamiento
[~,Result]= confusion.getMatrix(etiqueta_Train,eti_entre);
fprintf('Porcentaje de aciertos para dataset de entrenamiento es: %.2f%% \n
',Result.Accuracy*100);

 % Resultados de muestras de prueba
[c_matrix,Result]= confusion.getMatrix(etiqueta_Test,eti_prueba);
```

```
fprintf('Porcentaje de aciertos para dataset de prueba es: %.2f%% \n
',Result.Accuracy*100);
```

Las siguientes líneas del código son para realizar una malla que nos permita visualizar en la Figura 2.67 de mejor manera los datos de prueba clasificados y que ya se ha explicado en la MLP.

```
% Creación de malla para observar la frontera de decisión del clasificador
minimos = min(Xp);
maximos = max(Xp);
mallax = linspace(minimos(1),maximos(1),500);
mallay = linspace(minimos(2),maximos(2),500);
coordenadas = [mallax.' mallay.'];
[u, v] = meshgrid(mallax, mallay);
malla = [u(:)'; v(:)'];
mallaz = zscore(malla.');
mallaz = mallaz.';

% Evaluación la malla para graficar
[mallapre, mallaprese] = ProbarSVM(etiqueta_Test, mallaz, XY_Train,...
 modelo, ker, gamma, grado);

mallaprese = rescale(mallaprese, 0, 0.9);
sal_malla = mallapre - mallaprese;
sal_malla = reshape(sal_malla, size(u,1),size(u,2));

% Graficas de resultado de datos de prueba
       Graficar(eti_prueba, etiqueta_Test.', Xp.',sal_malla+2, coordenadas)
```

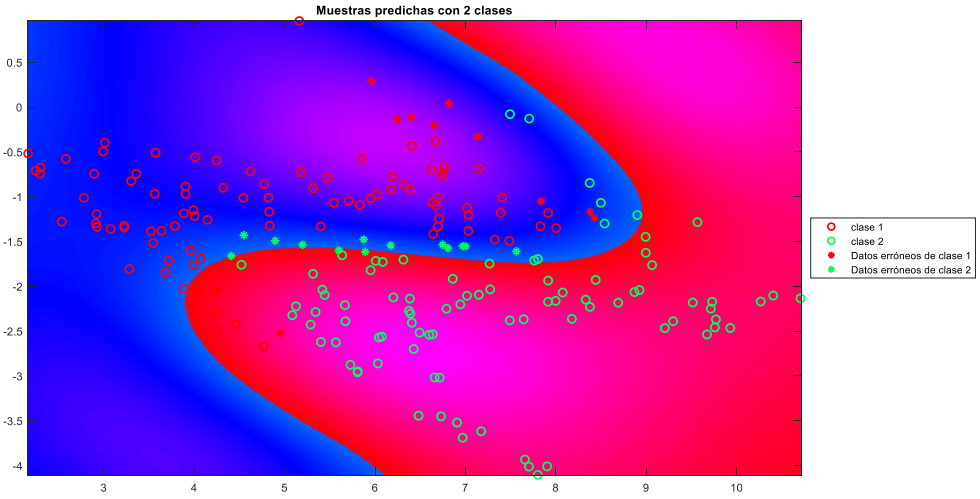


Figura 2.67 Resultados de clasificar datos con SVM

2.3.5.2. Función “ajustarsvm”

A continuación, se observan los elementos de la función. En su salida se tiene al modelo ajustado en la variable “`modelo`”, que consiste un tipo de dato tipo estructura donde se guardarán todas las variables necesarias para probarlo. Se verá las entradas de la función:

X: representa los datos a clasificar.

Kernel: el kernel con el cual se trabajará

y: representan a las salidas deseadas de los datos a clasificar.

C: representa la penalización cuando el algoritmo se equivoca en una muestra.

gamma: Es la constante entre cero y uno que se utilizan en el kernel de distribución gaussiana.

Grado: es la variable que se utiliza en el kernel polinómico.

```
function modelo = ajustarsvm(X, kernel, y, C, gamma, grado)
```

En las siguientes líneas en “n” se guardará el número de elementos de “y”, en “Y” se guardará una matriz diagonal cuadrada con los elementos del vector “y” en la diagonal principal como se puede observar en la Figura 2.68. La variable tol representa una tolerancia numérica para asegurar a convergencia de algoritmo.

```
n = numel(y);
Y = diag(y);
tol = 1e-4;
```

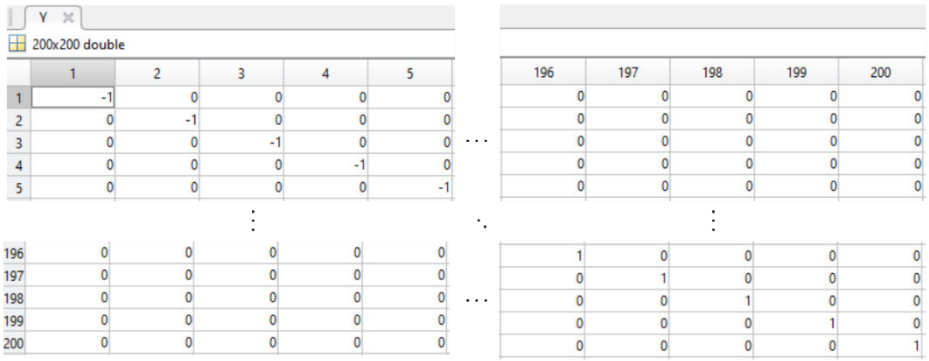


Figura 2.68 Matriz con la que se usará para en el algoritmo

En las siguientes líneas, se calcula el kernel en el que existen tres casos uno para kernel lineal otro para el kernel gaussiano y otro para el kernel polinómico, si se toma el kernel lineal se calcula de acuerdo con la Ecuación (**1.36**), si se escoge del kernel de base radial gaussiana se tiene con la Ecuación (**1.38**) antes de eso se calcula la norma entre dos vectores entrada en el kernel, y si es polinómico se calcula con la Ecuación (**1.37**). El resultado del cálculo se puede observar en la Figura 2.69, en donde se puede apreciar el resultado de la operación del Kernel.

```
switch kernel
    case 'lineal'
        K = X'*X ;
    case 'rbf'
        M = size(X,2);
        N = size(X,2);
        D = size(X,1);
        D2 = zeros(M,N);
        for d = 1:D
            D2 = D2 + (X(d,:)'*ones(1,N) - ones(M,1)*X(d,:)).^2 ;
        end
        K = exp(- gamma * D2);
```

```
    case 'poli'
        K = (X'*X+1).^grado;
End
```

K

200x200 double

	1	2	3	4	5	⋯	196	197	198	199	200
1	1	0.9275	0.8995	0.9318	0.0714	⋯	3.9377e-05	9.5681e-05	1.4625e-06	1.4687e-05	3.6771e-07
2	0.9275	1	0.9969	0.9957	0.1614	⋯	8.8914e-05	1.6425e-04	5.2623e-06	3.6399e-05	1.5003e-06
3	0.8995	0.9969	1	0.9942	0.1837	⋯	1.1647e-04	2.0552e-04	7.4942e-06	4.8678e-05	2.1903e-06
4	0.9318	0.9957	0.9942	1	0.1527	⋯	1.1995e-04	2.2768e-04	6.8777e-06	4.9292e-05	1.9553e-06
5	0.0714	0.1614	0.1837	0.1527	1	⋯	3.9414e-04	1.8713e-04	2.3175e-04	2.5286e-04	1.2212e-04
⋮						⋱			⋮		
196	3.9377e-05	8.8914e-05	1.1647e-04	1.1995e-04	3.9414e-04		1	0.7732	0.4855	0.9664	0.3124
197	9.5681e-05	1.6425e-04	2.0552e-04	2.2768e-04	1.8713e-04		0.7732	1	0.1597	0.6387	0.0822
198	1.4625e-06	5.2623e-06	7.4942e-06	6.8777e-06	2.3175e-04	⋯	0.4855	0.1597	1	0.6228	0.9476
199	1.4687e-05	3.6399e-05	4.8678e-05	4.9292e-05	2.5286e-04		0.9664	0.6387	0.6228	1	0.4355
200	3.6771e-07	1.5003e-06	2.1903e-06	1.9553e-06	1.2212e-04		0.3124	0.0822	0.9476	0.4355	1

Figura 2.69 Resultados de la función de kernel

Se usa la función de Matlab “quadprog” que devuelve un vector real como solución que está sujeta a todos los límites y restricciones locales. Por entradas se tiene:

`Y*K*Y`: Término objetivo cuadrático, especificado como una matriz real simétrica.

`ones(n,1)`: Término objetivo lineal, especificado como un vector real.

Los 2 siguientes términos no se incluyen, estos son el término objetivo lineal y las restricciones de desigualdad lineales especificadas por una matriz.

`y`: son las restricciones de desigualdad lineales especificadas por un vector.

`0`: son las restricciones de desigualdad lineales especificadas por un vector, pero no las incluimos para este caso.

`zeros(n,1)`: Límites inferiores, especificados como un vector o una matriz reales.

`C * ones(n,1)`: Punto inicial, especificado como un vector real.

`optimset('display', 'off', 'largescale', 'on')`: corresponde a las opciones de optimización, display off significa que no mostraremos ninguna salida.

Los “alfas” calculados como resultados de la función se pueden ver en la Figura 2.70.

```
% Programación cuadrática
alfa = quadprog(Y*K*Y, - ones(n,1), [], [], y, 0, zeros(n,1), C * ones(n,1), ...
                 [], optimset('display','off'));
```

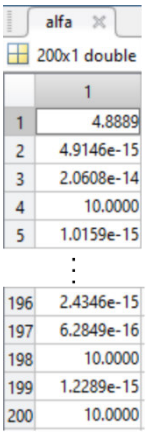


Figura 2.70 Vector de alphas

Se calculan los índices de los vectores de soporte, índices de los vectores cercanos al límite de decisión, vectores con valores mínimos y salidas que se plantean predecir. En la Figura 2.71 se planea

En la Figura 2.71 se muestran los vectores y variables de la estructura modelo a partir del algoritmo.

```
% Índices de los vectores en el límite de decisión
modelo.bndind = find(alfa > tol * C & alfa < (1 - tol) * C) ;
% Índices de los vectores de soporte
modelo.svind = find(alfa > tol * C) ;
% vector que tiene mínimos valores
modelo.alfa = alfa ;
% Salidas a predecir
modelo.alfay = Y * alfa ;
% Valor de compensación en SVM
modelo.b = mean(y(modelo.bndind) - modelo.alfay' * K(:,modelo.bndind)) ;
end
```

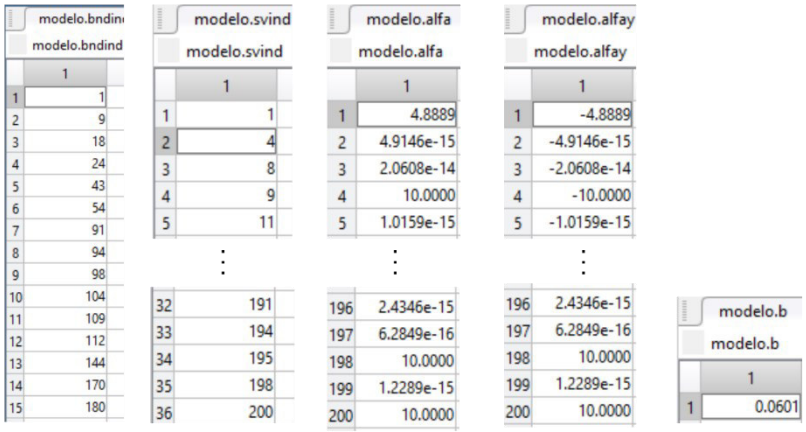


Figura 2.71 Valores correspondientes al modelo y que sirven para calcular las nuevas muestras.

2.3.5.3. Función “Probarsvm”

A continuación, se verán las salidas y entradas de la función. En las salidas se tiene la variable `ypredicho` que nos indica a qué clase pertenece el dato del que se quiere saber su clase y la salida `ypre` que es el valor que sale del algoritmo, es decir la salida sin etiqueta. También se presenta las siguientes salidas:

y: representan a las salidas deseadas del conjunto de entrenamiento.

`X_test`: representa los datos de prueba.

`modelo:` representa al modelo ya entrenado previamente.

`Kernel`: el kernel con el cual se trabajará

`C`: representa la penalización cuando el algoritmo se equivoca en una muestra.

`gamma`: Es la constante entre cero y uno que se utilizan en el kernel de distribución gaussiana.

`Grado`: es la variable que se utiliza en el kernel polinómico.

```
function [ypredicho, ypre] = ProbarSVM(y, X_test, X, modelo, kernel, gamma, grado)
```

Se crea un vector con los valores únicos por cada clase, nos servirá de referencia para determinar si se tiene que utilizar el algoritmo para mayor de dos clases o el de siempre.

```
clases = unique(y);
```

En esta parte del código se decide si se utilizará el código para solo dos clases o el que se tiene para 4 clases. Si se tiene más de 2 clases se aplica el algoritmo uno contra todos, para todas las clases, como se indica en la sección 1.3.2.2.4 se calcula los kernels como se muestra en la sección anterior con la diferencia que se hace de acuerdo con el número de clases que se tiene.

```
if length(clases) > 2
    ysalida = ones(length(X_test), length(clases));
    for i = 1: length(clases)
        switch kernel
            case 'l'
                K_test = X(:,modelo{1,i}.svind)' * X_test;
            case  'g'
                Xaux = X(:,modelo{1, i}.svind);
                M = size(Xaux,2);
                N = size(X_test,2);
                D = size(Xaux,1);
                D2 = zeros(M,N);
                for d = 1:D
                    D2 = D2 + (Xaux(d,:)'*ones(1,N) - ones(M,1)*X_test(d,:)).^2;
                end
                K_test = exp(- gamma * D2) ;
            case 'p'
                K_test = (X(:,modelo{1,i}.svind)' * X_test+1).^grado;
```

```
        end
```

Una vez que se ha calculado el kernel se calcula las salidas dadas por la Ecuación (**1.32**) para cada clase y las se ubica en una matriz.

```
        ysalida(:,i) = modelo{1,i}.alphay(modelo{1,i}.svind)' * K_test...
            + modelo{1,i}.b;
    End
```

Una vez que se tiene la matriz de salidas, se usan los máximos valores, donde ypre (Figura 2.72) que pertenece a la salida del algoritmo y la variable `I` las etiquetas de la salida, posteriormente se calcula la transpuesta de los vectores y se resta al índice 1 para que coincidan con las salidas que se pretende predecir.(Figura 2.73)

```
        [ypre, I] = max(ysalida,[],2);
        ypre = ypre.';
```

ypre

1x400 double

	1	2	3	4	5	6	7	8	9	10	11
1	2.7996	2.1956	-0.0612	0.3179	2.2761	2.0853	0.5285	3.7935	1.0105	1.2604	3.3651

Figura 2.72 Salida antes de etiquetar

```
        ypredicho = I.'-1;
```

ypredicho

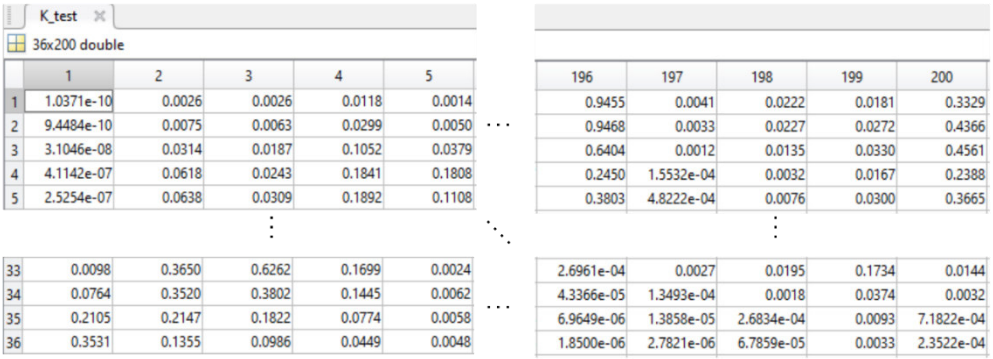
1x400 double

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	3	0	0	0	0	0	0	0	0	0

Figura 2.73 Salida después de etiquetar

Si solo se tiene 2 clases se procede a calcular el kernel de la misma manera en donde se creó el modelo. El resultado se puede ver en la Figura 2.74.

```
else
    switch kernel
    case 'l'
        K_test = X(:,modelo.svind)' * X_test;
    case  'g'
        Xaux = X(:,modelo{1, i}.svind);
        M = size(Xaux,2);
        N = size(X_test,2);
        D = size(Xaux,1);
        D2 = zeros(M,N);
        for d = 1:D
            D2 = D2 + (Xaux(d,:)'*ones(1,N) - ones(M,1)*X_test(d,:)).^2;
        end
        K_test = exp(- gamma * D2);
    case 'p'
        K_test = (X(:,modelo.svind)' * X_test+1).^grado;
```



K_test

36x200 double

	1	2	3	4	5	...	196	197	198	199	200
1	1.0371e-10	0.0026	0.0026	0.0118	0.0014	...	0.9455	0.0041	0.0222	0.0181	0.3329
2	9.4484e-10	0.0075	0.0063	0.0299	0.0050	...	0.9468	0.0033	0.0227	0.0272	0.4366
3	3.1046e-08	0.0314	0.0187	0.1052	0.0379		0.6404	0.0012	0.0135	0.0330	0.4561
4	4.1142e-07	0.0618	0.0243	0.1841	0.1808		0.2450	1.5532e-04	0.0032	0.0167	0.2388
5	2.5254e-07	0.0638	0.0309	0.1892	0.1108		0.3803	4.8222e-04	0.0076	0.0300	0.3665
⋮			⋮			⋱			⋮		
33	0.0098	0.3650	0.6262	0.1699	0.0024		2.6961e-04	0.0027	0.0195	0.1734	0.0144
34	0.0764	0.3520	0.3802	0.1445	0.0062	...	4.3366e-05	1.3493e-04	0.0018	0.0374	0.0032
35	0.2105	0.2147	0.1822	0.0774	0.0058		6.9649e-06	1.3858e-05	2.6834e-04	0.0093	7.1822e-04
36	0.3531	0.1355	0.0986	0.0449	0.0048		1.8500e-06	2.7821e-06	6.7859e-05	0.0033	2.3522e-04

Figura 2.74 Resultado de kernel

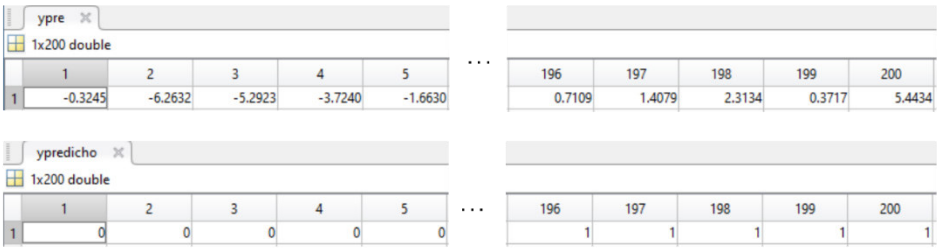
```
end
```

Se calcula la salida de del algoritmo de la siguiente manera:

```
ypre = modelo.alfay(modelo.svind)' * K_test + modelo.b;
ypredicho = ones(1,length(ypre));
```

Se crea la matriz donde irán las variables que ya han sido clasificadas. Luego se procede a etiquetar, si es que la salida es mayor a cero pertenece a la clase 1 pero si la salida sin etiquetar es mayor a cero pertenece a la clase 2. En la Figura 2.75 en el vector “ypre” se puede observar las salidas del algoritmo que toman varios valores entre positivos y negativos, mientras que en el vector “ypredicho” se puede observar las etiquetas ya clasificadas donde “cero” representa la clase 1 mientras que “uno” a la clase 2.

```
for j = 1:length(ypre)
if ypre(j) > 0, ypredicho(j) = 1 ; end
if ypre(j) < 0, ypredicho(j) = 0 ; end
end
```



ypre

1x200 double

	1	2	3	4	5	...	196	197	198	199	200
1	-0.3245	-6.2632	-5.2923	-3.7240	-1.6630	...	0.7109	1.4079	2.3134	0.3717	5.4434

ypredicho

1x200 double

	1	2	3	4	5	...	196	197	198	199	200
1	0	0	0	0	0	...	1	1	1	1	1

Figura 2.75 salida del algoritmo antes de etiquetar y salida del algoritmo después de etiquetar.

```
end
end
```

2.3.6. DECISIÓN TREE

A continuación, se explica la implementación del algoritmo Decision Tree cuya teoría se puede encontrar en la página 25. El propósito del algoritmo es crear un árbol de decisión similar al de la Figura 1.24.

Funciones ya descritas:

- Función “Graficar” (ya ha sido explicada en MLP)

- Función “Confusión” (ya ha sido explicada en MLP)
- La función “predicedistancia”, no es usada debido a la naturaleza del algoritmo, que en sí no predice distancia.

Funciones nuevas:

- Función “DTmain”, que es la función principal.
- Función “hacerArbol”, es la función que crea el árbol.
- Función “probarArbol”, es la función que crea el árbol.

2.3.6.1. Función “hacerArbol”

Primero se debe tomar en cuenta que es una **función recursiva**, es decir, esta se llama a sí misma para crear los árboles de decisión, por lo tanto, se tendrá un caso base para no tener que llamar a la función indefinidamente.

La función tiene como salida una estructura, que agrupa datos relacionados mediante contenedores denominados campos. Cada campo puede contener cualquier tipo de datos. Se puede acceder a los datos en un campo utilizando la notación “nombre_de_estructura.campo”. Esta será utilizada para probar nuevos datos, una vez que se haya formado la estructura, con el dataset muestras conocidas.

Debido a que este ejemplo es ligeramente diferente se va a separarlo en varios pasos estos son:

Paso 1: Inicialización de valores.

Paso 2: Definición de casos base.

Paso 3: Cálculo de la entropía de cada clase.

Paso 4: Hallar máxima entropía de cada atributo o dimensión.

Paso 5: Elección de dimensión que se utilizará para dividir.

Paso 6: División de datos en 2 grupos.

Paso 7: Creación de nueva rama con los nuevos grupos de datos.

Las entradas de la función son:

- `XY_Train`: muestras con etiquetas conocidas, representada en la Figura 2.76.
- `etiqueta_Train`: etiquetas conocidas, representada en la Figura 2.77.
- `err_nodo`: Porcentaje de muestras asignadas incorrectamente en un nodo.

```
function arbol = hacerArbol(XY_Train,etiqueta_Train,err_nodo)
```

Sus entradas tienen la forma:

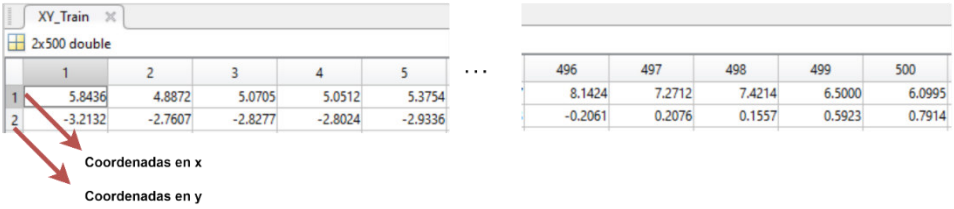


Figura 2.76 Entrada de la función árbol, se muestran los valores de los ejes en cada muestra.

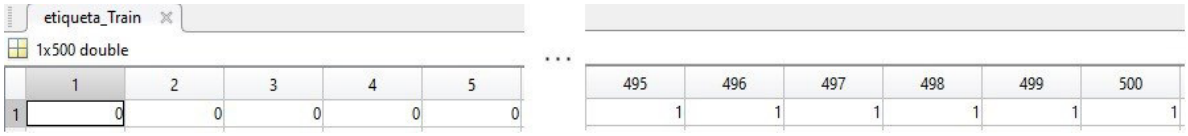


Figura 2.77 Etiqueta de cada muestra

Paso1: Inicialización de valores.

Se extrae número de muestras con la variable "`N_muestras`" y el número de dimensiones con la variable "`N_dim`", "N_dim" a la izquierda significa que el algoritmo solo está trabajando en 2 dimensiones, mientras que a la derecha "N_muestras" indicaando que se va a usar 500 muestras para crear el árbol.

```
[N_dim, N_muestras] = size(XY_Train);
```

```
N_dim =      N_muestras =

     2            500
```

Vector "`Eti_Unica`" con valores de etiquetas de diferentes clases, en eso consiste la función "unique" que devuelve los mismos datos que en `etiqueta_Train`, pero sin repeticiones. "`Eti_Unica`" está ordenado. En este caso se tiene 2 clases: 0 y 1.

```
Eti_Unica = unique(etiqueta_Train);
```

```
Eti_Unica =

     0     1
```

Se inicializa la variable **`arbol.dim`** de la estructura donde se guardarán las dimensiones de cada nodo. Mientras que en **`arbol.num_div`**, se guarda el valor división o umbral, en este caso se inicializa como infinito.

```
arbol.dim = 0;
arbol.num_div = inf;
```

Paso 2: Definición de casos base

Este caso base se da cuando el número de errores en el nodo es mayor a la cantidad de datos de entrada o la longitud de estos es 1, incluso si la longitud del número de clases es 1.

```
% Cuándo detenerse: si la dimensión es 1 o el número de ejemplos es pequeño
if ((err_nodo>N_muestras) || (N_muestras==1) || (length(Eti_Unica)==1))
```

También se hace un histograma almacenado en h donde, se cuenta el número de elementos del target. En este caso hist tiene 2 entradas:

`etiqueta_Train:` Contiene las etiquetas de una clase determinada.

`length(Eti_Unica):` nos dice en cuántos tipos de datos se va a clasificar el vector de targets.

`H:` es una salida expresada en vectores que nos indica el número de elementos de cada clase.

```
H = hist(etiqueta_Train, length(Eti_Unica));
```

Posteriormente, se calcula el índice de H donde hay mayor cantidad de elementos del objetivo.

```
[~, cy] = max(H);
```

Como es una hoja se deja en blanco Nf que significa el vector de los datos preparados para la división y el número que sirve de referencia para dividir los datos, que por lógica tampoco se va a utilizar.

```
arbol.Num_ele = [];
arbol.num_div = [];
```

Finalmente, en n nodos se guarda el índice donde se encuentran más elementos de una clase determinada. Se debe aclarar que dim es el campo que representa la dimensión que se ha escogido para cada muestra en la cual se harán comparaciones de sus medidas con los demás datos. Cuando dim es 1 significa que trabaja en eje x, si trabaja en dim 2 trabaja en eje “y”, en la Figura 2.78 se puede ver que la estructura se está inicializando para crear árbol de decisión.

```
arbol.nodos = Eti_Unica(cy);
```

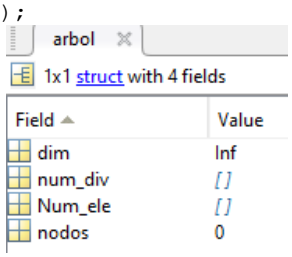


Figura 2.78 Inicialización de árbol

```
return
```

```
end
```

- **Paso 3: Cálculo de la entropía de cada clase**.

Para calcular la entropía en el nodo primero se calcula la probabilidad de que ocurra cada clase. En la Figura 2.79 Probabilidad de que ocurra cada clase., se observa cómo se guardan las probabilidades de que ocurra cada clase en la variable "P_clase"

```
% Probabilidad de que una muestra tome cierta clase
P_clase = zeros(length(Eti_Unica),1);

for i = 1:length(Eti_Unica)
    P_clase(i) = length(find(etiqueta_Train==Eti_Unica(i))) / N_muestras;
end
```

```
P_clase =

    0.5000
    0.5000
```

Figura 2.79 Probabilidad de que ocurra cada clase.

Para completar con el cálculo de la entropía se aplica la fórmula de Ecuación (**1.48**. En la Figura 2.80 se observa que en la variable "E_clase" se guarda la entropía calculada.

```
% Entropía de cada clase
E_clase = -sum(P_clase.*log2(P_clase));
```

```
E_clase =

     1
```

Figura 2.80 Entropía de cada clase.

El array "M_gan", guarda los valores de máxima ganancia de cada dimensión

```
% Para cada dimensión, se calcula la ganancia
M_gan = zeros(1, N_dim);
```

El array "num_div", guarda los valores de "x" y "y" de cada dimensión con los valores de cada ganancia.

```
num_div = zeros(1, N_dim);
```

Como se puede ver la variable `N_dim` significa el número de atributos o dimensiones de los datos, es decir que comenzamos a iterar entre las dimensiones.

```
for i = 1:N_dim
```

Se inicializa la variable data en donde van a estar todos los datos de la dimensión correspondiente, seguido se trata de ver los valores únicos que se presentan en la variable data, luego se calcula cuántos valores únicos hay con la variable Nbins.

```
    data = XY_Train(i,:);
    Ud = unique(data);
    Nbins = length(Ud);
```

Se inicializa la matriz donde se guardará las probabilidades

```
    %Patrón continuo
    Prob = zeros(length(Eti_Unica), 2);
```

Primero se ordena los datos de menor a mayor de los datos de entrada, posteriormente se ordena para los targets. En la Figura 2.81 se observa el vector con datos ordenados de menor a mayor, en la Figura 2.82 para no perder el hilo se guarda en otro vector los índices y en la Figura 2.83 se reordenan las etiquetas correspondientes de los datos para no perder hilo.

```
% Ordenar patrones
[dato_ordenado, indices] = sort(data);
```

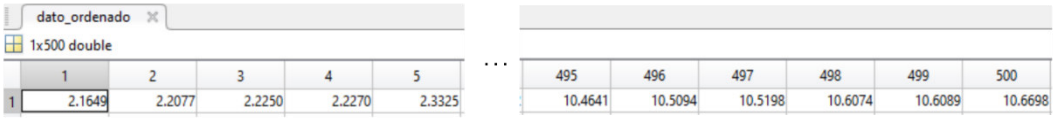


Figura 2.81 Vector con valores ordenados

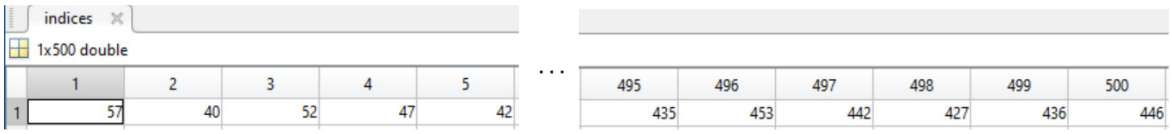


Figura 2.82 Índices valores ordenados.

```
eti_ordenada = etiqueta_Train(indices);
```

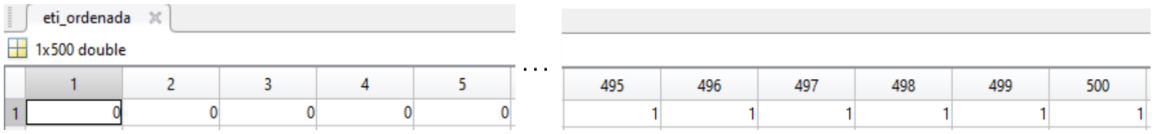


Figura 2.83 Etiquetas del árbol

Se calcula la ganancia de información para los datos de entrada, en la Figura 2.84 se inicializa la ganancia de información.

```
% Calcular división de salida
GananciaI    = zeros(1, N_muestras-1);
```

Figura 2.84 Ganancia de información en cada punto

Se toma todos los datos de entrada y se itera, si cumple con la condición en la que la muestra de siguiente es de diferente clase a la actual entra a hacer los cálculos siguientes.

```
for j = 1:N_muestras

  if eti_ordenada(j)~= eti_ordenada(j+1)
```

Con la matriz de probabilidades se forma un histograma con los valores únicos de las salidas, en la columna 1 se tendrá el número veces que se tiene una clase hasta la `j`

iteración (Figura 2.85), mientras que en la segunda columna se tiene un histograma con los valores que van desde `j` hasta el final de los datos ordenados (Figura 2.86).

```
Prob(:, 1) = hist(eti_ordenada(1:j) , Eti_Unica);
```

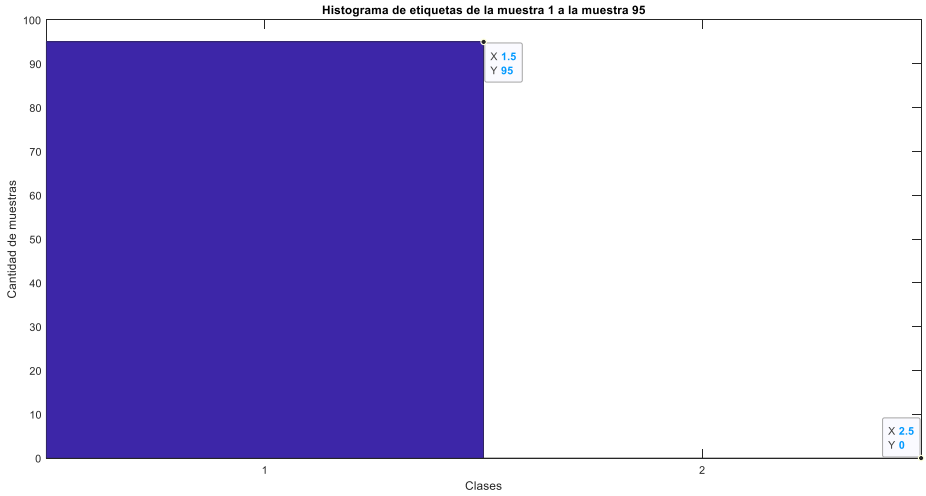


Figura 2.85 Resultado de función “hist” de Matlab

```
Prob(:, 2) = hist(eti_ordenada(j+1:end) , Eti_Unica);
```

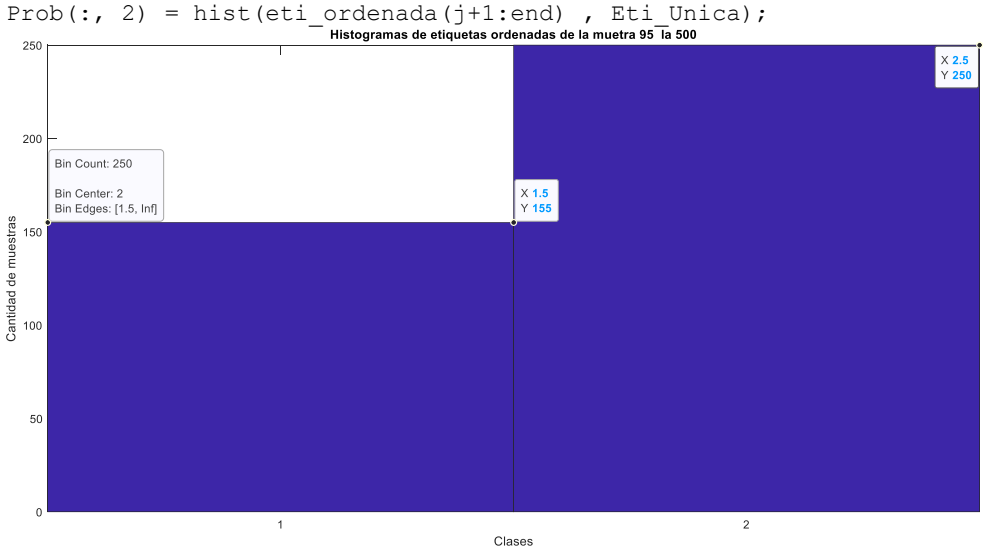


Figura 2.86 Resultado de función “hist” de Matlab

La matriz queda de la siguiente manera.

```
Prob =

    95   155
     0   250
```

Cantidad de muestras de clase 1 pertenecientes menores que 2.1649 ← 95 155 → Cantidad de muestras de clase 1 pertenecientes mayores que 2.1649

Cantidad de muestras de clase 2 pertenecientes menores que 2.1649 ← 0 250 → Cantidad de muestras de clase 2 pertenecientes mayores que 2.1649

El valor de 2.1649 proviene de que representa el valor en el eje “x” un dato cualquiera del conjunto de datos conocidos, en este caso.

Se calcula la probabilidad de que un elemento sea mayor o menor del margen

```
sum_p = sum(Prob)/N_muestras;
```

```
sum_p =

    0.1900    0.8100
```

Probabilidad que muestras sean menores que 2.1649 ← 0.1900 0.8100 → Probabilidad que muestras sean mayores que 2.1649

Se calcula una matriz de probabilidades simplemente dividiendo la matriz para la cantidad de datos conocidos totales.

```
Prob = Prob/N_muestras;
```

```
Prob =

    0.1900    0.3100
         0    0.5000
```

Repetimos la `Ps` en una matriz `"P_aux"` las veces correspondientes al número de clases `(Eti_Unica)` que existen. Para poder sumar las

```
P_aux = repmat(sum_p, length(Eti_Unica), 1);
```

Cuando hay casos en los que las probabilidades son cero se va a tener problemas en los cálculos con los logaritmos, por lo tanto, a cada valor de la matriz se suma el menor número que puede ser almacenado en Matlab, este se llama “eps” y su valor es de 2.2204e-16, por lo tanto, se acercará a cero y permitirá calcular mejor las ecuaciones que se tiene con algoritmos.

```
P_aux = P_aux + eps;
```

Se aplica la fórmula de la entropía para calcular la con las nuevas probabilidades (Figura 2.87).

```
E_nueva = sum(-Prob.*log2(eps+Prob./P_aux));
```

```
E_nueva =

   -0.0000    0.7775
```

Figura 2.87 Nueva Entropía

Se aplica la fórmula de la ganancia de información para cada valor en el vector.

```
GananciaI.
        GananciaI(j) = E_clase - sum(E_nueva.*sum_p);
```

En la Figura 2.88 se observa que la muestra 95 como una candidata potencial que sirva como umbral.

Ganancial

1x499 double

	94	95	96	97	98	99	100	101	102
1	0	0.3702	0	0	0	0	0	0	0

Figura 2.88 La mayor ganancia de información en muestra 95

```
end
```

```
End
```

Paso 4: Hallar máxima entropía de cada atributo o dimensión

Se busca la mayor ganancia de información de cada dimensión en "`M_gan(i)`", con su respectivo índice guardado en "`s`". En la Figura 2.89 se puede observar que al final de la iteración se guarda la mayor ganancia de información con su respectivo índice.

```
[M_gan(i), s] = max(GananciaI);
```

```
M_gan =                    s =

    0.5605          0       228
```

Figura 2.89 Se guarda el valor y su respectivo índice para una nueva iteración

Se guarda en la variable, las muestras que servirán como punto de inflexión para dividir los datos, tomando en cuenta el índice de la máxima ganancia.

```
    num_div(i) = dato_ordenado(s);
End
```

Se puede observar a la muestra con la máxima ganancia :

```
num_div =

    6.2413   -1.4843
```

Es decir, los números donde se tiene la máxima ganancia para cada dimensión en "x" sería 6.2413 y en "y" se tiene a -1.4843

Se puede observar que la variable "s" es el número donde se guarda el número de mayor ganancia de Información es:

```
s =

   231
```

Paso 5: Elección de dimensión que se utilizará para dividir.

La variable `dim` se guarda la dimensión con la que se trabajará para dividir el árbol, en este caso se elige la dimensión "y" debido a que dim es igual a 2.

```
[~, dim] = max(M_gan);
```

```
dim =

     2
```

Se actualiza en dims las dimensiones con las que se está trabajando actualmente.

```
dims = 1:N_dim;
```

```
dims =

     1     2
```

En la Figura 2.90 se observa cómo se guarda en la estructura en la variable dim la dimensión con la que se trabajará. En este caso es la dimensión 2 que representa la “y”.

```
arbol.dim = dim;
```

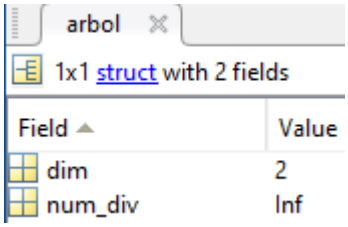


Figura 2.90 Dimensión con la que se trabajará

En la Figura 2.91 se observa cómo se guarda en el vector “`Un_ele`” los datos que se usa para guardar el nodo.

```
% Se busca si alguna muestra tiene el mismo valor que otra
Un_ele = unique(XY_Train(dim,:));
```

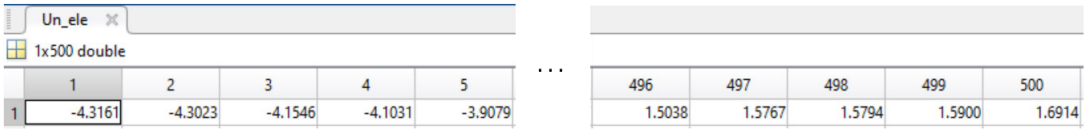


Figura 2.91 Vector donde se guardan los valores de nodo

Se calcula el número de elementos de la dimensión con la que se está trabajando.

```
Nbins = length(Un_ele);
```

Se adjunta en la variable “`Un_ele`” de la estructura árbol los datos de la dimensión que se escoge para el nodo (Figura 2.92).

```
arbol.Num_ele = Un_ele;
```

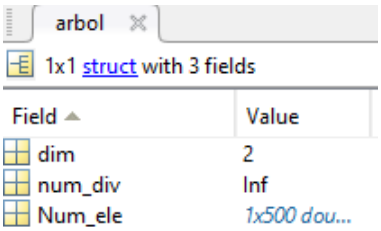


Figura 2.92 Se guarda la dimensión que se escogió para dividir y elementos que se están dividiendo.

Se guarda en la variable de la estructura el número clave con que separa el nodo principal en nodos secundarios o en hojas, este nodo corresponde al número -1.4843 (Figura 2.93), del eje y.

```
% se agrega a la estructura el número de división entre un conjunto y otro
arbol.num_div  = num_div(dim);
```

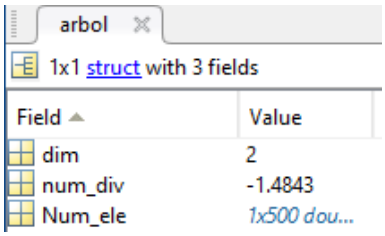


Figura 2.93 Número clave con que separa el nodo principal en nodos secundarios o en hojas

En la Figura 2.94 se puede ver que ya se da forma a este Nodo principal, el "500" que se ve significa el número de muestras con las que se está formando el nodo, en la anterior figura en dim se tiene el valor 2 que equivale al "eje y", el número -1.483 es el que tenía la máxima ganancia por lo tanto se toma como referencia para dividir este conjunto de entrenamiento en 2 grupos.

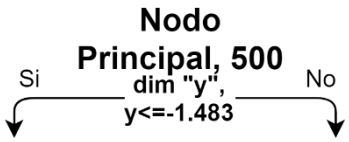


Figura 2.94 Nodo principal representado gráficamente

Paso 6: División de datos en 2 grupos.

Si los datos son continuos, se separa los datos y se coloca en dos vectores, datos por encima del número que se elige para dividir el nodo como el vector `indices_may` (Figura 2.95) y datos por debajo del valor que se elige para dividir el nodo como el vector `indices_men`. (Figura 2.96)

```
indices_men = find(XY_Train(dim,:) <= num_div(dim));
```

Figura 2.95 Datos por menore del número de división

```
indices_may = find(XY_Train(dim,:) > num_div(dim));
```

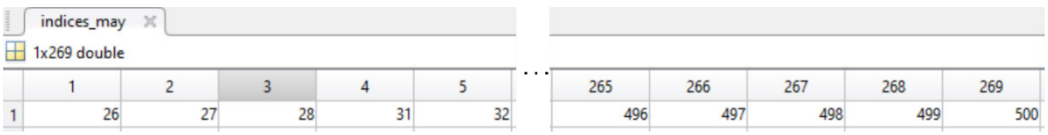


Figura 2.96 Datos por encima del número de división

En los siguientes nodos se puede observar que se dividen un conjunto de datos de 231 muestras para nodo 1 y 269 para nodo 2 (Figura 2.97).

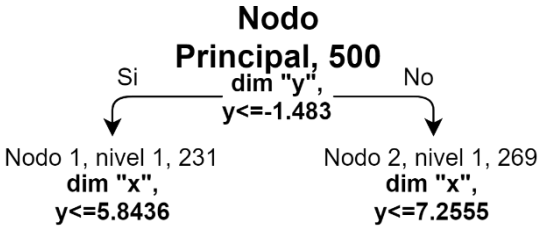


Figura 2.97 Árbol con 2 niveles

Paso 7: Creación de nueva rama con los nuevos grupos de datos.

Se procede a llamar la función `hacerArbol`, con los datos que se escoge para llamarla de manera recursiva hasta crear este árbol.

El nodo principal divide en 2 nodos cuando la dim 2 que pertenece al eje y de los datos, y se toma la muestra que contiene el valor en el eje “y” con más alta ganancia de información en este caso este número se llama num_div y se divide en 2 nodos más, como se ve en la figura un nodo para valores menores a “num_div” se crea el primer nodo y para valores mayores a se crea el segundo, de manera recursiva hasta llegar hasta el caso base.(Figura 2.98)

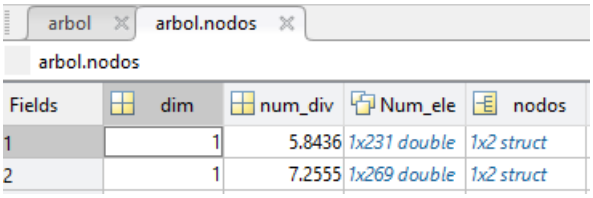


Figura 2.98 matrices donde se guardan los modelos

```
arbol.nodos(1) = hacerArbol(XY_Train(dims, indices_men),...
    etiqueta_Train(indices_men), err_nodo);
```

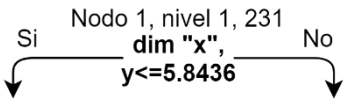


Figura 2.99 Nodo 1, nivel 1

En la Figura 2.99 se ve que el nodo 1 del nivel 1, este es similar al nodo principal. Se llama a la misma función y esta se repetirá hasta llegar hasta el caso base.

```
arbol.nodos(2) = hacerArbol(XY_Train(dims, indices_may),...
  etiqueta_Train(indices_may), err_nodo);
```

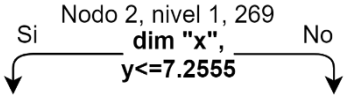


Figura 2.100 Nodo 2, nivel 1

Como se puede notar en la Figura 2.100 se ve que el nodo 2 del nivel 1, este es similar al nodo principal. Se llama a la misma función y esta se repetirá hasta llegar hasta el caso base.

A continuación, en la Figura 2.101 se ve como queda el árbol, finalmente.

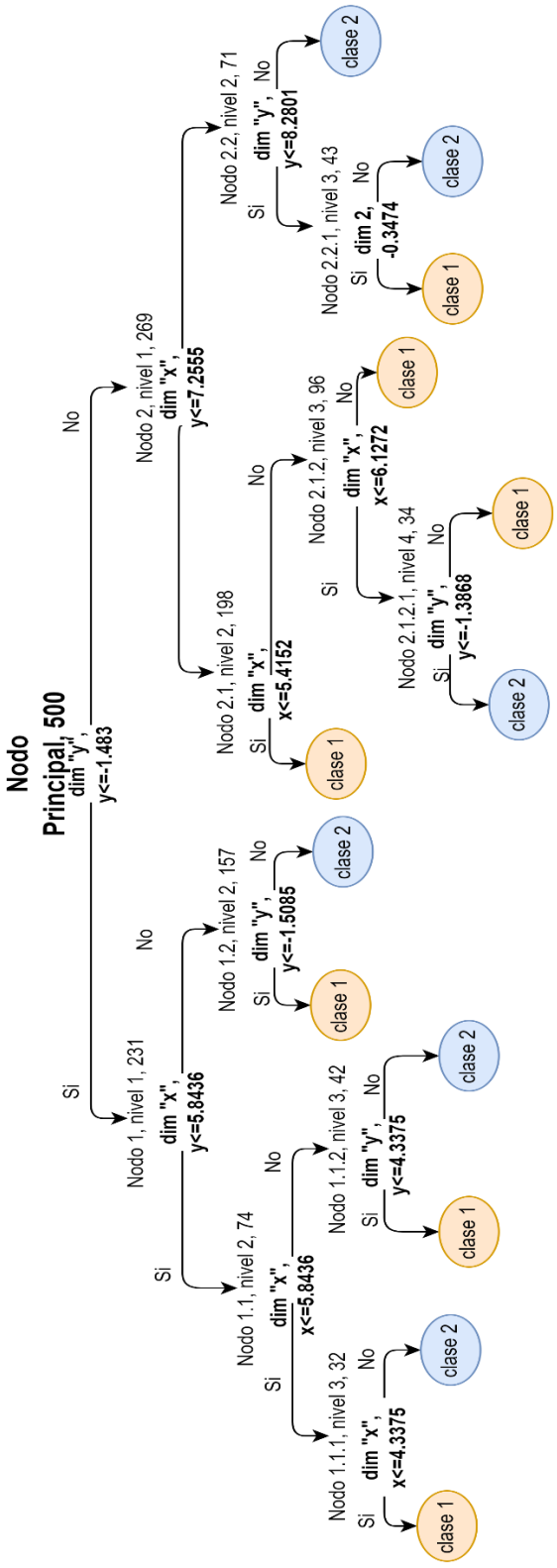


Figura 2.101 Árbol de decisión final, realizado con los datos de entrenamiento.

2.3.6.2. Función “probarArbol”

La función tiene como salida un vector con las salidas que predijo el algoritmo.

Como entradas tiene:

- `XY_Test:` muestras de prueba.(Figura 2.102)

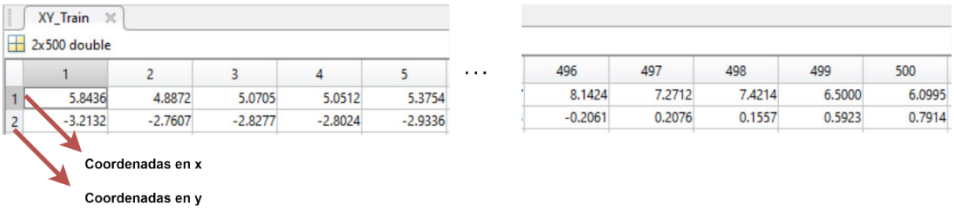


Figura 2.102 Muestras de prueba

- `indices:` vector de índices de las muestras de prueba, estos se usarán como referencia para clasificar las muestras. (Figura 2.103)

Figura 2.103 Etiquetas de muestras de prueba

- `arbol:` Estructura con los datos necesarios para la clasificación de los datos. (Figura 2.104)

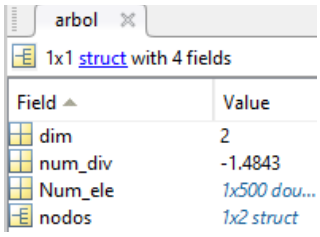


Figura 2.104 Modelo creado previamente para clasificar los nuevos datos

```
ypred = probarArbol(XY_Test, indices, arbol)
% clasificar recursivamente usando el árbol
```

Se comienza inicializando el vector en donde se guardarán los valores de salida que se está dispuestos a predecir.

```
ypred = zeros(1, size(XY_Test,2));
```

Cuando el campo “dim” de la estructura árbol llega a cero, al vector “ypred” con índices de las muestras de prueba se le asigna el campo “nodos”, significa que el árbol ya no se divide más, En la Figura 2.105 si bien no es evidente se nota que el árbol ya no se divide más.

```
if (arbol.dim == 0)
    % Llegó al final del árbol
    ypred(indices) = arbol.nodos;
    return
```

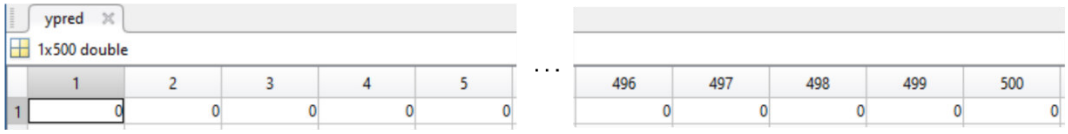


Figura 2.105 etiquetas predichas por el algoritmo

En la Figura 2.106 se observa que se llega al final de una rama se puede para todos los índices que se ha asignado se tiene:

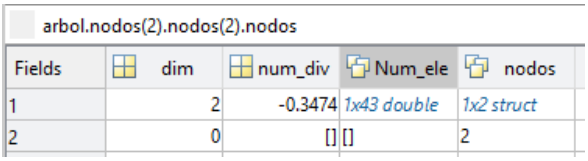


Figura 2.106 Ramas del árbol

En la Figura 2.107, se ve que en cierta rama elementos mayores que 8.2801 serán etiquetados con la clase 2. En la Figura 2.108 se ve de manera gráfica cómo se dividen los nodos.

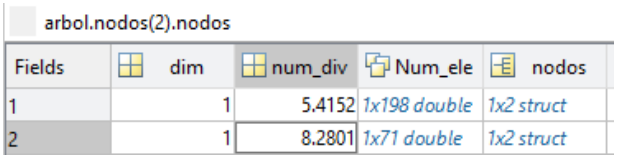


Figura 2.107 Nodo 2 del nivel 2

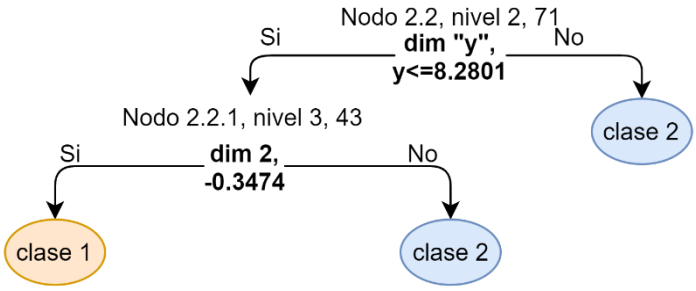


Figura 2.108 Nodo 2 del nivel 3 con su respectiva gráfica.

```
end
```

Primero se ve en las dimensiones en las que se va a trabajar, posteriormente se busca en la estructura la dimensión escogida .

```
% Si este no es el último nivel del árbol, entonces:
% Primero, encuentre la dimensión en la que se va a trabajar
dim = arbol.dim;
dim =

     2

dims = 1:size(XY_Test,1);
```

```
dims =

     1     2
```

En `in` se guarda los índices de las muestras de prueba menores a -1.4843, debido a que se eligió la dimensión “y” donde dim es igual 2. En la Figura 2.109 se observa un vector de los datos para donde están disponibles para dividir el árbol.

```
in = indices(XY_Test(dim, indices) <= arbol.num_div);
```

num_div -1.4843

in — 1x231 double

	1	2	3	4	5	...	227	228	229	230	231
1	1	2	3	4	5		458	460	461	462	463

Figura 2.109 Indices que se planeas buscar

Se guarda en `ypred` las etiquetas resultantes de este árbol y se vuelve a llamar a la función su propiedad recursiva. En la Figura 2.110 aunque no sea muy evidente en la primera iteración ya está clasificando los datos.

```
ypred   = ypred + probarArbol(XY_Test(dims, :), in, arbol.nodos(1));
```

ypred — 1x500 double

	1	2	3	4	5	...	496	497	498	499	500
1	0	0	0	0	0		0	0	0	0	0

Figura 2.110 Salida del árbol en la segunda iteración

En `in` se vuelve a hacer lo mismo pero esta vez para valores mayores a -1.4843. En la Figura 2.111 se observa un vector de los datos para donde están disponibles para dividir el árbol.

```
in = indices(XY_Test(dim, indices) >  arbol.num_div);
```

in — 1x13 double

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	7	14	23	289	291	293	294	307	309	310	316	318

Figura 2.111 Indices para buscar valores mayores al índice previamente creado.

Se guardan en `ypred` las etiquetas resultantes del árbol y se vuelve a llamar a la función debido que esta es recursiva. En la Figura 2.112 aunque no sea muy evidente en la primera iteración ya está clasificando los datos.

```
ypred   = ypred + probarArbol(XY_Test(dims, :), in, arbol.nodos(2));
```

ypred — 1x500 double

	1	2	3	4	5	...	496	497	498	499	500
1	0	0	0	0	0		0	0	0	0	0

Figura 2.112 Salida del árbol en la segunda iteración

2.3.6.3. Función “DTmain”

Se limpia la pantalla y otras variables que no tengan que ver con este algoritmo.

```
% DTmain
clc; clear, close all;
%% Cargar Dataset de prueba
```

Se carga muestras con etiquetas conocidas, y se coloca en la variable `datos_entrenar`.

```
load('Entrenamiento.mat');
datos_entrenar = train2c500;
```

Se separa los datos en una matriz que contienen los atributos, en este caso son las coordenadas de los puntos.

```
XY_Train = datos_entrenar(:,1:2);

XY_Train = XY_Train.';
```

Se separa los datos en un vector de salidas esperadas o etiquetas de cada muestra.

```
etiqueta_Train = datos_entrenar(:,3);
etiqueta
```

```
%% Cargar Dataset de prueba
```

Se carga muestras con etiquetas conocidas, y se coloca en la variable `datos_ prueba`.

```
load('Prueba.mat');
datos_prueba = test4c;
```

Se separa los datos en una matriz de muestras.

```
XY_Test = datos_prueba(:,1:2);
xT = XY_Test.';
```

Se separa los datos en un vector de salidas esperadas.

```
etiqueta_Test = datos_prueba(:,3);
etiqueta_Test = etiqueta_Test.';
```

Se va a utilizar una técnica de poda, con esta se supone que cuando llega a un nodo con menos del 4% de muestras esta ya no clasifica más, para no crear árboles muy robustos y que sobreajusten datos.

```
err_nodo = 4;
```

Se inicializan variables Ni como número de dimensiones y M como el número de muestras. Luego se calcula el número de muestras que permite este error. “Ni” representa el número de clases y “M” el número de datos de entrenamiento. En “err_nodo” se muestra el resultado del cálculo para determinar cuántas muestras de error se admiten en el algoritmo.

```
%% Entrenamiento
[Ni, M] = size(XY_Train);
```

```
Ni =     M =

     2     500
```

```
err_nodo = err_nodo*M/100;
```

```
err_nodo =

    20
```

Se llama a la función para crear los datos con las entradas que ya se define antes.

```
% Construyendo árbol recursivo
disp('Construyendo árbol')
```

Como entradas tiene:

- `XY_Train:` muestras con etiquetas conocidas.
- `etiqueta_Train:` etiquetas conocidas.
- `err_nodo:` Porcentaje de muestras asignadas incorrectamente en un nodo.

Y como salidas tiene:

- `arbol:` Es una estructura que va a guardar todas las variables necesarias para clasificar nuevas muestras (Figura 2.113)

```
arbol = hacerArbol(XY_Train, etiqueta_Train, err_nodo);
```

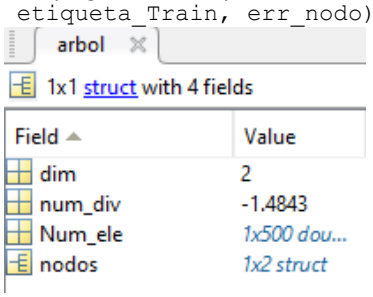


Figura 2.113 Estructura creada para clasificar nuevos datos.

`dim:` indica la dimensión que se escoge como referencia para dividir el nodo

`num_div:` número escogido como referencia para dividir el conjunto de datos en menores a num_div y mayores a num_div.

`Num_ele:` indica el número de elementos en ese nodo

`Nodos:` es la estructura donde se encuentran localizados los subnodos en los que se divide el nodo principal, se debe recalcar que cuando llegan a una hoja ya no se dividen más.

Se debe aclarar que la imagen anterior es la representación del nodo principal.

A continuación, se presentan los subnodos del nodo raíz, esto se presenta de manera recursiva. Los subnodos de la primera división del árbol se pueden apreciar en la Figura 2.114.

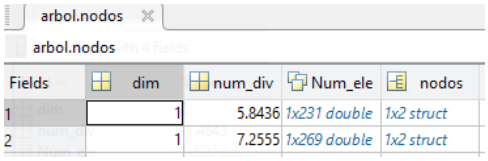


Figura 2.114 Subnodos del nodo raíz

Se prueba el conjunto de entrenamiento para comprobar en un futuro un sobre ajuste. Para ello se usa la función “probarArbol”.

Como entradas tiene:

- `XY_Train:` muestras de prueba.
- `1:size(XY_Train,2):` vector de índices de las muestras de prueba, estos se usarán como referencia para clasificar las muestras.

`arbol:` Estructura con los datos necesarios para la clasificación de los datos.

Y como salidas tiene:

`predtrain:` es un array de las etiquetas predichas por el dataset de entrenamiento (Figura 2.115).

```
% Probar Entrenamiento
predtrain = probarArbol(XY_Train, 1:size(XY_Train,2), arbol);
```

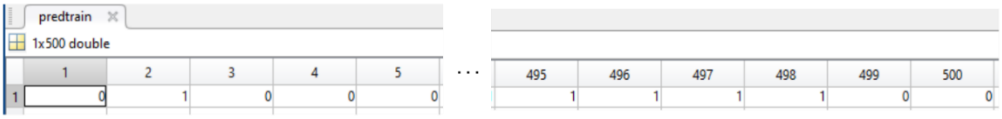


Figura 2.115 Etiquetas predichas por este árbol creado

Se llama a la función getmatrix de la clase confusión para ver el porcentaje de muestras que han sido acertadas por el algoritmo.

```
[~,Result]= confusion.getMatrix(etiqueta_Train, predtrain);
 fprintf('Porcentaje de aciertos para dataset de entrenamiento es: %.2f%% \n ',
Result.Accuracy*100);
```

Se llama la función de prueba para el dataset de prueba. Los resultados se ven en Figura 2.116 con la variable “tar_pre”.

```
disp('Clasificación de muestras de prueba')
tar_pred = probarArbol(xT, 1:size(xT,2), arbol);
```

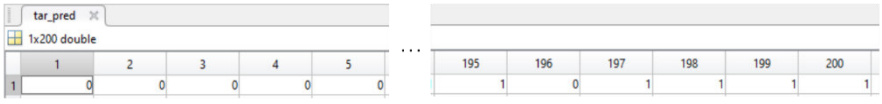
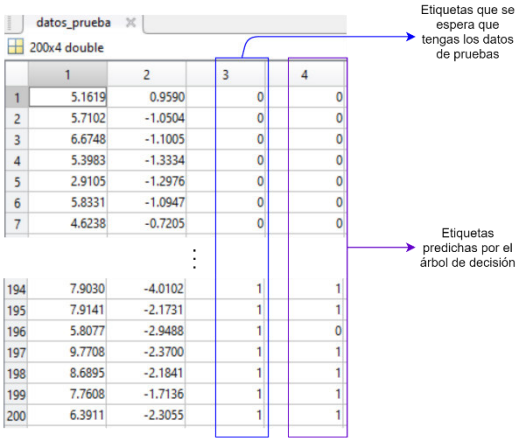


Figura 2.116 Muestras predichas con dataset de prueba

En la Figura 2.117 se puede ver el parecido entre las etiquetas de datos de prueba que se espera obtener y las que se predicen con el árbol creado anteriormente.



	1	2	3	4
1	5.1619	0.9590	0	0
2	5.7102	-1.0504	0	0
3	6.6748	-1.1005	0	0
4	5.3983	-1.3334	0	0
5	2.9105	-1.2976	0	0
6	5.8331	-1.0947	0	0
7	4.6238	-0.7205	0	0
⋮				
194	7.9030	-4.0102	1	1
195	7.9141	-2.1731	1	1
196	5.8077	-2.9488	1	0
197	9.7708	-2.3700	1	1
198	8.6895	-2.1841	1	1
199	7.7608	-1.7136	1	1
200	6.3911	-2.3055	1	1

Figura 2.117 Comparación entre etiquetas predichas y etiquetas reales

Se obtiene los mínimos y los máximos valores para graficarlos correctamente.

```
minimos = min(XY_Test);
maximos = max(XY_Test);
```

Se crea una malla con los mínimos y máximos de los datos.

```
% Creación de malla para observar la frontera de decisión del clasificador
mallax = linspace(minimos(1),maximos(1),100);
mallay = linspace(minimos(2),maximos(2),100);
coordenadas = [mallax.' mallay.'];
[u, v] = meshgrid(mallax, mallay);
malla = [u(:)'; v(:)'];
```

Se crea la malla característica para representar los datos predichos. El resultado se muestra las muestras predichas de la malla se observa en la Figura 2.118, mientras que en la Figura 2.119 se observa la gráfica resultante con un fondo entre clases diferentes a demás algoritmos debidos a que las salidas de este algoritmo son plenamente discretas.

```
ymalla = probarArbol(malla, 1:size(malla,2), arbol);
```

	1	2	3	4	5	…	9995	9996	9997	9998	9999	10000
1	0	0	0	0	0		1	1	1	1	1	1

Figura 2.118 Muestras predichas en cada punto de la malla

```
ymalla = reshape(ymalla, size(u,1),size(u,2));

Graficar(tar_pred, etiqueta_Test, xT, ymalla +… length(unique(etiqueta_Test)),
coordenadas);
```

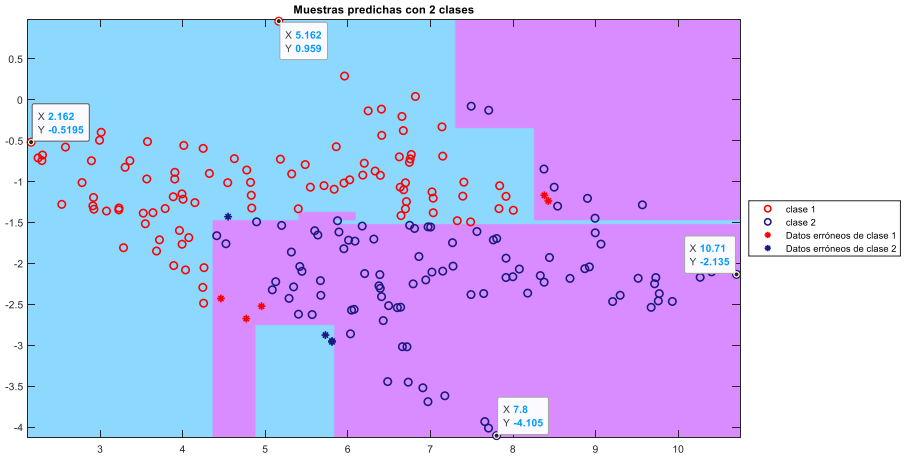


Figura 2.119 Muestras predichas con dos clases por parte del algoritmo DT

Los resultados de las métricas con el dataset de pruebas se calculan en las siguientes líneas en la Figura 2.120 se aprecian todas las métricas.

```
[~,Result]= confusion.getMatrix(etiqueta_Test, tar_pred);
fprintf('Porcentaje de aciertos para dataset de prueba es: %.2f%% \n ',...
    Result.Accuracy*100);
```

```
Métricas:
Exactitud: 0.955000
Precisión: 0.959596
Sensibilidad: 0.950000
Especificidad: 0.960000
Medida F1: 0.954774
Porcentaje de aciertos para dataset de prueba es: 95.50%
```

Figura 2.120 Resultados de las métricas para dataset de prueba.

Se grafica la matriz de confusión (Figura 2.121), para ver los resultados de una manera más intuitiva.

```
figure
cm = confusionchart(etiqueta_Test,tar_pred,'RowSummary','row-normalized',
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
```

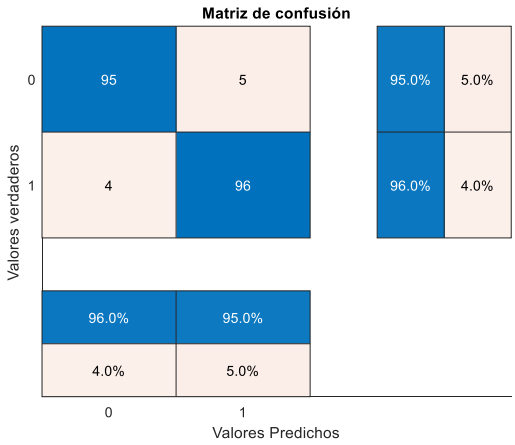


Figura 2.121 Matriz de confusión

2.3.7. NAIVE BAYES

A continuación, se explicará la implementación del algoritmo Naive Bayes cuyo código está basado en la teoría de la página 31 .Consta de las siguientes funciones para facilitar la comprensión de este.

- Función “Graficar” (ya ha sido explicada)
- Función “Confusión” (ya ha sido explicada)
- Función “alejamiento” (ya ha sido explicada)
- El código principal no tiene la ayuda de otras funciones.

Se va a basar el código en los 4 pasos, adicionalmente agregaremos nuevos pasos:

- Paso 1: cargar datos.
- Paso 2: Se calcula la probabilidad de que ocurra cada clase.
- Paso 3: Se calcula la media y la varianza de los atributos por clase.
- Paso 4: Se prueba con las muestras de Prueba en la función “normpdf”.
- Paso 5: Se busca la clase a la que pertenece
- Paso 6: Repetir pasos 4 y 5 en cuanto se necesite
- Paso 7 Realizar la malla para hacer intuitivos los gráficos
- Paso 8: Graficar.

Programa principal

Paso 1: cargar datos.

Primero se carga los datos de entrenamiento y en una matriz se carga los atributos de las muestras y en un array se carga las etiquetas de cada muestra.

```
clc, clear, close all;
% Ingreso de datos
load('Entrenamiento.mat');
datos_entrenar = train2c500;
XY_Train = datos_entrenar(:,1:2);
etiqueta_Train = datos_entrenar(:,3)+1
```

Se carga los datos de prueba con el mismo formato que se carga el dataset de entrenamiento.

```
% Cargar datos
load('Prueba.mat');
datos_prueba = test2c;
X_test = datos_prueba(:,1:2);
```

```
y_test = datos_prueba(:,3)+1;
```

Se calcula el número de clases con la que se va a trabajar.

```
n_etiquetas = length(unique(etiqueta_Train));
```

```
n_etiquetas =

     2
```

En la variable "N_conocido" se guarda el número de muestras conocidas en este caso serán 500, mientras que en "Dim_conocido" se guarda el número de atributos o dimensiones del conjunto de datos en este caso serán 2, "x" y "y".

```
[N_conocido, Dim_conocido] = size(XY_Train);
```

```
N_conocido =

   500
```

Se inicializan las matrices donde se guardarán las variables pertenecientes a la media y a la varianza con las mismas dimensiones.

```
Med = zeros(n_etiquetas, Dim_conocido);


var = zeros(n_etiquetas, Dim_conocido);
```

Paso 2: Se calcula la probabilidad de que ocurra cada clase.

Se calcula las probabilidades de cada clase, en este caso como cada clase tiene la misma cantidad de muestras será equiprobable.

```
% Calculando prioridades
% prev:  probabilidad previa
prev = [];
for j=1: n_etiquetas
    prev = [prev; length(yTrain(y_train == j))/N];
end
```

```
prev =

    0.5000 ⟶ Probabilidad de clase 1
    0.5000 ⟶ Probabilidad de clase 2
```

Paso 3: Se calcula la media y la varianza de los atributos o dimensiones por clase.

Se calcula la media y la desviación estándar de los valores de los atributos de cada clase. Antes se calcula la varianza y luego la desviación estándar. Se debe recordar que la media (Figura 2.122) se calcula de acuerdo con la Ecuación (**1.53**) y la desviación estándar(Figura 2.123) con la Ecuación (**1.54**). La varianza no está definida en alguna ecuación anterior pero

```
% Calculando media y desviación estándar
for j=1: n_etiquetas
    tmp = X_train(y_train==j,:); % todos los datos de la clase j
   for d=1: Dim_conocido
       Med(j,d) = sum(tmp(:,d))/length(y_train(y_train == j));
       var(j,d) = (sum((tmp(:,d)-Med(j,d)).^2)/length(y_train(y_train ==
j)));
   end
end
```

```
Med =

    5.2420   -0.8783
    7.5586   -1.9343
```

Media de clase 1 en eje x ← 5.2420 | Media de clase 1 en eje y → -0.8783

Media de clase 2 en eje x ← 7.5586 | Media de clase 2 en eje y → -1.9343

Figura 2.122 Media en cada eje según la clase

```
desv = var.^(1/2);
```

```
desv =

    1.6509    0.8430
    1.6380    0.8505
```

Desviación Estándar de clase 1 en eje x ← 1.6509 | Desviación Estándar de clase 1 en eje y → 0.8430

Desviación Estándar de clase 2 en eje x ← 1.6380 | Desviación Estándar de clase 2 en eje y → 0.8505

Figura 2.123 Desviación estándar Media en cada eje según la clase

Se va a probar primero el conjunto de entrenamiento.

```
probs = [];
for i=1:length(y_train)
    ejemPro = [];
```

Se ubica la probabilidad previa, significa la probabilidad de que la muestra pertenezca a una clase determinada.

```
    for j=1: n_etiquetas
        clasep = prev(j);
```

```
clasep =

    0.5000
```

Paso 4: Se prueba con las muestras de Prueba en la función "normpdf".

Representa la función de probabilidad normal, tiene la siguiente sintaxis:

y = normpdf(x,mu,sigma)

Donde:

La salida "y" es la probabilidad de un suceso según la media y desviación estándar dada, en este caso es la probabilidad de que la muestra x_k dada la clase C_i, y en este caso se presenta como Pxc.

La entrada “x” es el valor de la muestra que se quiere sacar la probabilidad dada la función de probabilidad de la clase.

La entrada “mu”, representa la media de la clase que se planea representar, en este caso es MC_i.

La entrada “sigma”, representa la desviación estándar en este caso es σ.

Matemáticamente se expresa de la siguiente manera:

$$Pxc_{k,i} = \frac{1}{\sqrt{2\pi{\sigma_i}^2}} e^{-\frac{(x_k-\mu_i)^2}{2{\sigma_i}^2}} \quad (2.1)$$

Donde:

$Pxc_{k,i}$: es una matriz que contiene las probabilidades de cada valor de las muestras evaluados en las n clases encontradas.

Para sacar la probabilidad a posteriori o la probabilidad de la clase dada la muestra se multiplica la probabilidad de que sea la clase p dada por la probabilidad con la Ecuación (**2.1**), también se puede usar la función como vimos anteriormente.

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)} \quad (2.2)$$

Donde:

$P(A_i)$: es la probabilidad de que ocurra A_i

$P(B|A_i)$: es la probabilidad de que ocurra B dado que haya ocurrido A_i.

$P(A_i|B)$: es la probabilidad de que ocurra A_i dado que haya ocurrido B.

```
        for d=1: Dim_conocido
            clasep = clasep * normpdf(XY_Train(i,d),Med(j,d),desv(j,d));
        end
```

```
clasep =

    0.1131
```

Se guarda en una vector que contiene las probabilidades que ocurra la clase que se está evaluando dada una muestra.

```
        ejemPro = [ejemPro clasep];

    end
```

```
ejemPro =

    0.0012    0.0107
```

Se guarda el vector en una matriz de probabilidad que contiene las probabilidades de que la muestra pertenezca a la cantidad de clases que se tiene en el conjunto de entrenamiento (Figura 2.124).

```
    probs = [probs; ejemPro];

end
```

probs

500x2 double

	1	2
1	0.0012	0.0107
2	0.0046	0.0094
3	0.0039	0.0104
4	0.0042	0.0105
5	0.0029	0.0118
⋮		
496	0.0089	0.0068
497	0.0117	0.0024
498	0.0113	0.0028
499	0.0093	5.6187e-04
500	0.0070	2.2598e-04

Figura 2.124 Matriz de con las probabilidades de que las muestras pertenezcan a una clase u otra.

Paso 5: Se busca la clase a la que pertenece

Se multiplica $Pxc_{k,i}$ por C_i. debido a que el denominador en el teorema de Bayes será igual para todos los casos, para clasificar las muestras se tiene:

$$E_p = argmax(Pxc_{k,i} * C_i) \quad (2.3)$$

Donde:

E_p: es la etiqueta predicha.

En la función siguiente se guardará en el vector ind_max los índices en los que están almacenado las máximas probabilidades y estas corresponden a las etiquetas de clases que predice el algoritmo (Figura 2.125).

```
[~,ind_max] = max(probs,[],2);
```

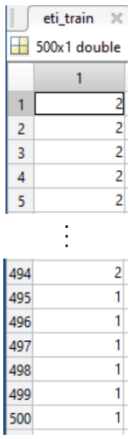


Figura 2.125 Etiquetas predichas

Se calcula el porcentaje de aciertos del dataset de prueba.

```
[~,Result]= confusion.getMatrix(y_train, predtrain);
fprintf('Porcentaje de aciertos para dataset de entrenamiento es: %.2f%% \n
',...Result.Accuracy*100);
```

Paso 6: Repetir pasos 4 y 5 en cuanto se necesite

Ahora se calcula las etiquetas de cada muestra con un conjunto de pruebas diferente al anterior, y se aplica el mismo algoritmo anterior.

```
% calcular probabilidades de prueba
probs = [];
for i=1:length(y_test)
    ejemPro = [];
    for j=1:K
        clasep = prev(j);
        for d=1:D
             clasep = clasep * normpdf(XY_Test(i,d),Med(j,d),desv(j,d));
        end
        ejemPro = [ejemPro clasep];
    end
    probs = [probs; ejemPro];
end
[probeti, eti] = max(probs,[],2);
```

Paso 7: realizar la malla con los pasos 4 y 5 para hacer intuitivos los gráficos

Se calculan los mínimos y máximos de las muestras en cada dimensión.

```
%% Resultados
% Creación de malla para observar la frontera de decisión del clasificador
minimos = min(XY_Test);
maximos = max(XY_Test);
```

Se crea un array de 200 puntos para la dimensión x.

```
mallax = linspace(minimos(1),maximos(1),100);
```

Se crea un array de 200 puntos para la dimensión y.

```
mallay = linspace(minimos(2),maximos(2),100);
```

Se agrupan en una matriz los datos de las coordenadas.

```
coordenadas = [mallax.' mallay.'];
```

La función `meshgrid` devuelve coordenadas de cuadrícula 2-D basadas en las coordenadas contenidas en los vectores `mallax` y `mallay`.

```
[u, v] = meshgrid(mallax, mallay);
```

Se vuelve a convertir en una matriz `nx2`, para que cada punto de la malla que se crea sea evaluado por el algoritmo de clasificación y etiquetarlo según corresponda.

```
malla = [u(:)'; v(:)'];
mallaz = malla.';
```

Se evalúa los puntos de la malla creada.

```
probs = [];
for i=1:length(mallaz)
    ejemPro = [];
    for j=1:K
        clasep = prev(j);
        for d=1:D
            clasep = clasep * normpdf(mallaz(i,d),Med(j,d),desv(j,d));
        end
        ejemPro = [ejemPro clasep];
    end
    probs = [probs; ejemPro];
end
[m_prob, m_eti] = max(probs,[],2);
```

En el vector, `mproeti` se puede observar que hay valores muy bajos, entonces se escalan esos valores a 0.1 y a 0.8, para que al momento de graficar se pueda observar el gráfico de una manera más intuitiva.

```
mallaprese = rescale(m_prob,0.1,0.8);
```

En el vector `mallaprese` se tiene simplemente los valores escalados, si se grafica ahora no se tendrá presente las zonas a las que pertenece cada clase, por lo tanto, se suma la etiqueta para que en el gráfico se pueda apreciar de mejor manera donde se encuentra cada muestra.

```
eti_malla = m_eti + mallaprese;
```

Ahora se agrupan las etiquetas sumadas a cada valor de salida escalado, formando una malla con los valores de salida que se crea para poder ver una gráfica más intuitiva.

```
eti_malla = reshape(eti_malla, size(v,1),size(v,2));
```

Paso 7: Graficar.

En la función graficar se tiene como entradas a:

`Eti`: que representa el etiquetado del algoritmo.

`y_test`: que representa a las etiquetas reales de las muestras de prueba.

`X_test`: representa a las muestras de prueba

`Ymalla`: representa la malla para graficar los puntos de cada clase, se suma 1 para que las muestras de prueba contrasten con los puntos de la malla.

`Coordenadas`: representan los puntos que se van a graficar.

En la Figura 2.126 se puede observar los resultados de la malla en conjunto con la clasificación de los datos. Posteriormente tenemos en la Figura 2.127 la matriz de confusión.

```
% Graficas de resultado de datos de prueba
Graficar(eti, y_test, X_test.',Ymalla+1, coordenadas)
```

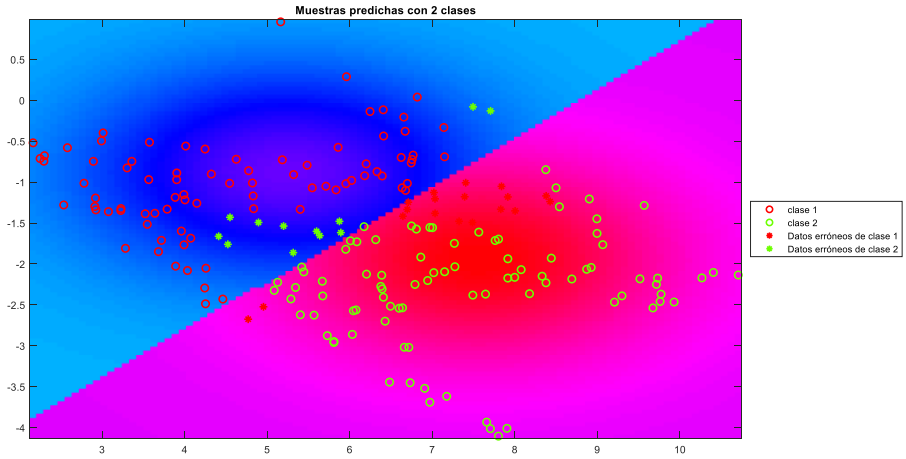


Figura 2.126 Resultados de muestras predichas por algoritmo Naive Bayes.

```
figure
cm = confusionchart(y_test,eti,'RowSummary','row-normalized',...
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
[~,Result]= confusion.getMatrix(y_test, eti);
fprintf('Porcentaje de aciertos para dataset de prueba es: %.2f%% \n ',...
    Result.Accuracy*100);
```

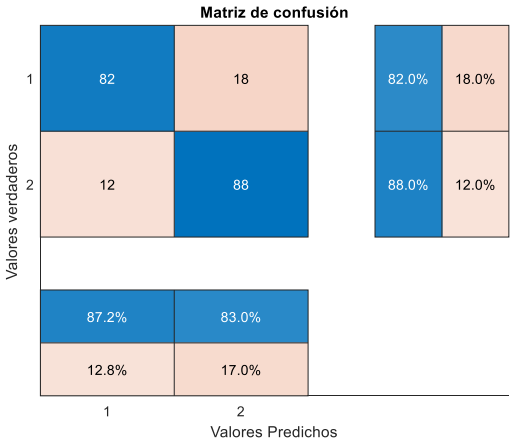


Figura 2.127 Matriz de confusión con datos predichos por algoritmo Naive Bayes

2.3.8. K-NEAREST NEIGHBOR

A continuación, se explica el código implementado en el algoritmo k-NN cuya teoría se encuentra en la página 34, básicamente se pretende clasificar una muestra como se indica en la Figura 1.29. El código consta de las siguientes figuras.

- Función “Graficar” (ya ha sido explicada)
- Función “Confusión” (ya ha sido explicada)
- Función “alejamiento” (ya ha sido explicada)
- Función “mainKNN”, que es el programa principal del algoritmo.
- Función “knn”, es la función que etiqueta las muestras de prueba.

Donde las salidas de las funciones son:

`nmax`: se refiere al número muestras de la etiqueta predicha estuvieron cerca de la muestra que se quiere predecir la etiqueta

`vpredicho`: es la etiqueta predicha.

Y sus entradas son:

`k`: número de vecinos más cercanos.

`xentrenamiento`: muestras de entrenamiento.

`tentrenamiento`: etiquetas de muestras de entrenamiento.

`xprueba`: muestras de prueba.

```
[nmax, eti_predicha] = knn(k, XY_Train, etiqueta_Train, XY_Test)
```

2.3.8.1. Función “knn”

Este algoritmo es relativamente simple por lo tanto se tendrá 5 pasos entre ellos están:

Paso 1. Cargar los datos.

Paso 2. Calcular distancia de cada muestra con etiqueta conocida a la muestra de prueba.

Paso 3. Buscar k-vecinos más cercanos.

Paso 4. Contar etiquetas de cada muestra más cercana.

Paso 5. Escoger clase con más etiquetas en el grupo de los k-NN

```
function [nmax, eti_predicha] = knn(k, XY_Train, etiqueta_Train, XY_Test)

% función del algoritmo knn
% ENTRADAS:
% k: Número de vecino más cercanos
% XY_Train: (NxD) datos con etiquetas conocidas; N es el número de
% muestras y D es la dimensionalidad de cada punto de datos
% eti_conocido: etiquetas de entrenamiento
% XY_Test: (MxD) datos de prueba;
% SALIDAS:
% eti_predicha: las etiquetas predichas basadas en el algoritmo k-NN
% nmax: número de muestras de la clase elegida
```

Paso 1. Cargar los datos.

En la Figura 2.128 se pueden ver que sus entradas son las siguientes:

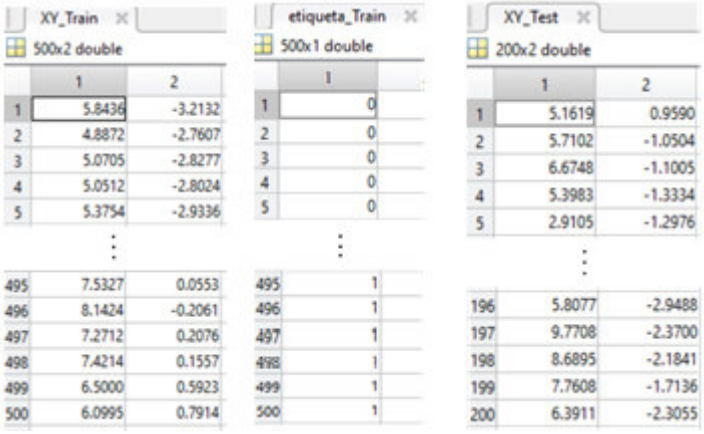


Figura 2.128 Datos a cargar

Las siguientes condiciones son necesarias para el funcionamiento de la función, si se tiene menor a 4 argumentos entonces nos sale un error, debido a que existen insuficientes argumentos.

```
if nargin < 4
    error('Muy pocos argumentos')
end
```

Los datos con etiquetas conocidas y los datos de prueba deben tener las mismas dimensiones para ingresar a la función.

```
if size(XY_Train,2) ~= size(XY_Test,2)
    error('Los datos deben tener la misma dimensionalidad');
end
```

Se recomienda elegir número impares para elegir el número de vecinos más cercanos.

```
if mod(k,2) == 0
    error('Elija impar para reducir posibilidad de empate.');
end
```

Se inicializa cada variable, matriz o vector.

```
%Inicializar
```

En este vector se guardarán las etiquetas predichas por el algoritmo.

```
eti_predicha = zeros(size(XY_Test,1),1);
```

Se guardará el número de vecinos más cercanos para asegurarnos de que pertenece a una clase u otra.

```
nmax = zeros(size(XY_Test,1),1);
```

Matriz donde se guarda las distancias de cada punto al punto que se planea etiquetar.

```
distancia = zeros(size(XY_Test,1),size(XY_Train,1)); distancia euclidiana
```

Índice donde se guardan los k-vecinos más cercanos

```
indice = zeros(size(XY_Test,1),size(XY_Train,1));
```

Matriz donde se guardan más cercanos de cada elemento del que se planea predecir la etiqueta.

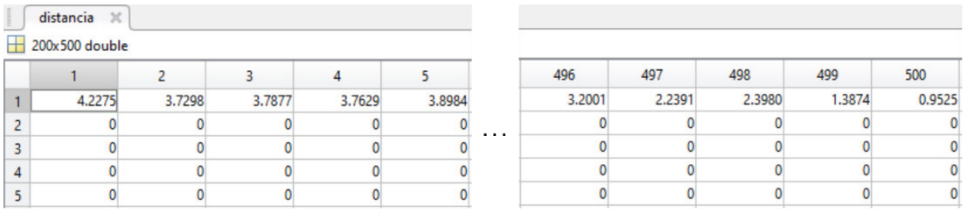
```
kNN = zeros(size(XY_Test,1),k); %k vecinos más cercanos para probar la
muestra
```

Paso 2. Calcular distancia de cada muestra con etiqueta conocida con la muestra de prueba.

Se calcula la distancia de cada punto de prueba con cada punto de etiqueta conocida en una matriz por lo tanto se tendrá una matriz 200x500. Luego se ordenará la matriz desde la distancia más corta a la más larga.

```
for pi = 1:size(XY_Test,1)
    for pj = 1:size(XY_Train,1)
% Calcular y almacenar distancias euclidianas clasificadas con los índices
correspondientes.
        distancia(pi, pj) = sqrt(sum((XY_Test(pi,:)-XY_Train(pj,:)).^2));
```

En la Figura 2.129 se puede observar cómo se calculan distancias desde la primera muestra de prueba a cada uno de los datos usados como referencia.



	1	2	3	4	5	...	496	497	498	499	500
1	4.2275	3.7298	3.7877	3.7629	3.8984		3.2001	2.2391	2.3980	1.3874	0.9525
2	0	0	0	0	0		0	0	0	0	0
3	0	0	0	0	0		0	0	0	0	0
4	0	0	0	0	0		0	0	0	0	0
5	0	0	0	0	0		0	0	0	0	0

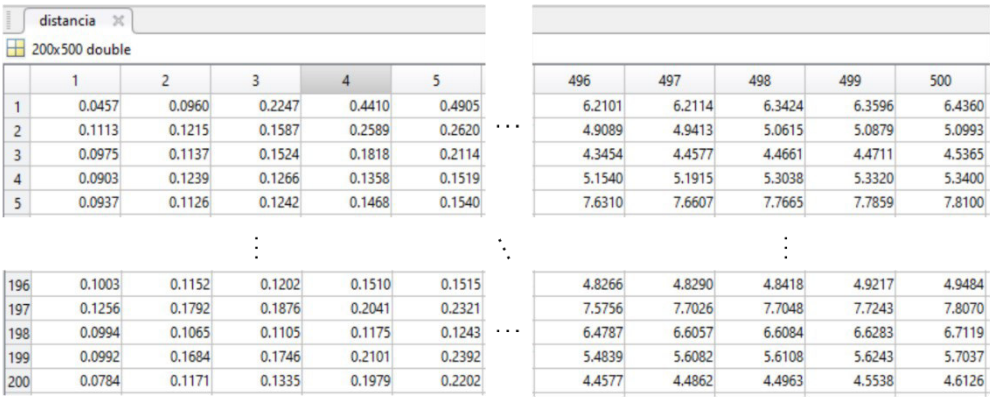
Figura 2.129 Distancia al primer punto de prueba a todos los puntos de dataset de entrenamiento

```
End
```

Paso 3. Buscar k-vecinos más cercanos.

Sen Figura 2.130 se observa cómo se ordena los datos de menor distancia de los puntos de prueba a puntos con etiquetas conocidas.

```
[distancia(pi,:),indice(pi,:)] = sort(distancia(pi,:));
```



	1	2	3	4	5	...	496	497	498	499	500
1	0.0457	0.0960	0.2247	0.4410	0.4905		6.2101	6.2114	6.3424	6.3596	6.4360
2	0.1113	0.1215	0.1587	0.2589	0.2620		4.9089	4.9413	5.0615	5.0879	5.0993
3	0.0975	0.1137	0.1524	0.1818	0.2114		4.3454	4.4577	4.4661	4.4711	4.5365
4	0.0903	0.1239	0.1266	0.1358	0.1519		5.1540	5.1915	5.3038	5.3320	5.3400
5	0.0937	0.1126	0.1242	0.1468	0.1540		7.6310	7.6607	7.7665	7.7859	7.8100
⋮						⋱			⋮		
196	0.1003	0.1152	0.1202	0.1510	0.1515		4.8266	4.8290	4.8418	4.9217	4.9484
197	0.1256	0.1792	0.1876	0.2041	0.2321		7.5756	7.7026	7.7048	7.7243	7.8070
198	0.0994	0.1065	0.1105	0.1175	0.1243		6.4787	6.6057	6.6084	6.6283	6.7119
199	0.0992	0.1684	0.1746	0.2101	0.2392		5.4839	5.6082	5.6108	5.6243	5.7037
200	0.0784	0.1171	0.1335	0.1979	0.2202		4.4577	4.4862	4.4963	4.5538	4.6126

Figura 2.130 Matriz con datos ordenados de menor distancia mayor

Además, en la Figura 2.131 se puede ver como se guardan los índices por cada punto de prueba.

indice

200x500 double

	1	2	3	4	5	⋯	496	497	498	499	500
1	233	232	248	241	235		435	442	446	436	427
2	132	120	133	116	111		435	442	436	446	427
3	158	148	153	140	161		42	47	52	40	57
4	349	103	101	105	353		453	442	436	427	446
5	31	38	28	27	26		453	442	436	427	446
⋮						⋱			⋮		
196	272	9	269	263	276		245	243	436	446	249
197	411	422	421	414	401		42	40	47	52	57
198	385	387	386	384	399	⋯	42	40	47	52	57
199	392	400	406	157	409		42	47	40	52	57
200	319	314	313	334	335		52	245	243	57	249

Figura 2.131 Matriz de índices de menor distancia a mayor

```
end
```

```
% Encontrar la k más cercana para cada punto de datos de los datos de prueba
```

En la matriz "kNN" se guardan los índices de los k vecinos más cercanos de cada punto (Figura 2.132).

```
kNN=indice(:,1:k);
```

kNN

200x3 double

	1	2	3
1	233	232	248
2	132	120	133
3	158	148	153
4	349	103	101
5	31	38	28
⋮			
196	272	9	269
197	411	422	421
198	385	387	386
199	392	400	406
200	319	314	313

Figura 2.132 Índices de los k vecinos más cercanos

Paso 4. Contar etiquetas de cada muestra más cercana.

Posteriormente se realiza una votación, para elegir la clase a la que pertenece cada punto del conjunto de prueba.

```
% Votación
for i = 1:size(kNN,1)
```

Entre los kNN se busca las etiquetas de los datos conocidos que están más cercanos a las muestras de prueba.

```
    opciones = unique(etiqueta_Train(kNN(i,:)'));
```

Se inicializará la variable "vmax" para almacenar el número de muestras que determinan la clase a la que pertenece una muestra.

```
vmax = 0;
```

En la Figura 2.133 se guardan de izquierda a derecha, el número de etiquetas con las variables más cercanas, en "vmax" número de muestras con variables más cercanas y en "etiquetamax", la etiqueta ganadora.

```
etiquetamax = 0;
```

```
opciones =    vmax =    etiquetamax =

     0             0             0
```

Figura 2.133 Variables inicializadas

```
for j = 1:length(opciones)
```

En "L" se guarda el número de muestras por cada clase que se encuentran más cercanas a cada muestra de prueba.

```
L = length(find(etiqueta_Train(kNN(i,:)')==opciones(j)));
```

En este caso las 3 muestras más cercanas tendrán pertenecerán a la clase1, "L" es una variable auxiliar con cierto número de muestras.

```
L =

     3
```

Si el nuevo L es mayor entonces la nueva etiqueta de la muestra será la del valor de "L", y el número de elementos más cercanos de la misma clase se guardará en "vmax".

```
if L > vmax
   etiquetamax = opciones(j);
   vmax = L;
```

```
etiquetamax =    vmax =

     0             3
```

Finalmente se guardará los valores de "vmax" en un vector "nmax" que contiene los valores de la cantidad de muestra que fueron necesarias para etiquetar cada dato de prueba. Y en "eti_predicha" se guardará las etiquetas predichas correspondientes a cada muestra de prueba. En la Figura 2.134 se puede observar el vector en donde se guarda con qué número de votos ha ido ganando cada clase en las muestras de prueba. En la Figura 2.135 se muestra cada muestra de prueba con la etiqueta ganadora o etiqueta predicha.

```
nmax(i) = vmax;
```

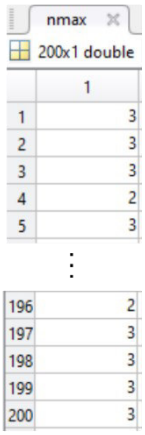


Figura 2.134 Número votos con el que se clasificó para cada muestra

```
eti_predicha(i) = etiquetamax;
```

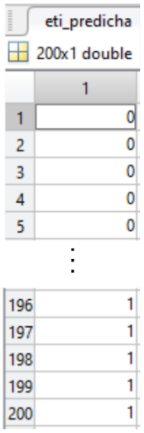


Figura 2.135 Etiqueta predicha por el algoritmo

```
end
```

2.3.8.2. Función “mainKNN”

```
%% K-NN
%% Datos de entrenamiento
```

Se cierran todas las ventanas y se limpian los datos en pantalla.

```
clc, clear; close all;
```

Se carga los datos de entrenamiento.

```
load('Entrenamiento.mat');
```

Se carga en la matriz “`datos_entrenar`” los datos con las etiquetas conocidas, que serán usados para predecir los demás datos de prueba.

```
datos_entrenar = train4c500;
```

Se separa en las coordenadas de cada punto conocido y en sus respectivas etiquetas.

```
XY_Train = datos_entrenar (:,1:end-1);
```

```
etiqueta_Train = datos_entrenar (:,end);
```

Se va a usar los 3 vecinos más cercanos, entonces los inicializamos en la función.

```
k = 3;
```

Se usa la función explicada anteriormente para evaluar los datos conocidos como se puede ver, pero con la intención de ver si hay un sobreajuste por parte del algoritmo de clasificación. No se tiene el número de muestras necesarias para clasificar la muestra porque no se necesita en este caso, solo se quiere evaluar el porcentaje de muestras clasificadas correctamente. En la Figura 2.136 se pueden ver las etiquetas predichas del conjunto llamado “Train500”, que es el conjunto con etiquetas conocidas.

```
[~, eti_pre_train] = knn(k, XY_Train, etiqueta_Train, XY_Train);
```

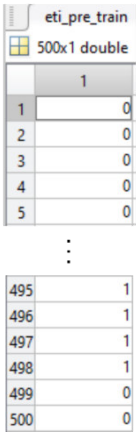


Figura 2.136 Etiqueta predicha

A continuación, se va a ver las muestras que ha predicho el algoritmo para sus muestras conocidas en porcentaje.

```
[~,Result]= confusion.getMatrix(etiqueta_Train, eti_pre_train);
fprintf('Porcentaje de aciertos para dataset de entrenamiento es: %.2f%% \n
',Result.Accuracy*100);
```

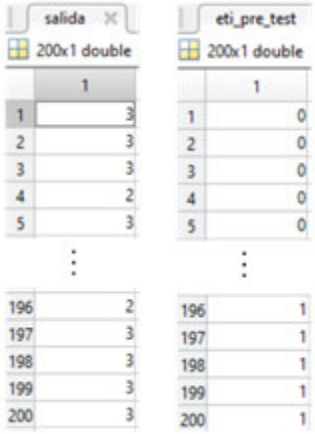
Se carga los datos de prueba, son diferentes a los datos anteriores.

```
%% datos de prueba
load('Prueba.mat');
prueba = test4c;
```

Se separa en las coordenadas de cada punto conocido y en sus respectivas etiquetas como se aprecia en Figura 2.137, mientras que en la Figura 2.138 se aprecia el mismo caso para el conjunto malla.

```
XY_Test = prueba(:,1:end-1);
etiqueta_Test = prueba(:,end);
```

```
[salida, eti_pre_test] = knn(k, XY_Train, etiqueta_Train, XY_Test);
```

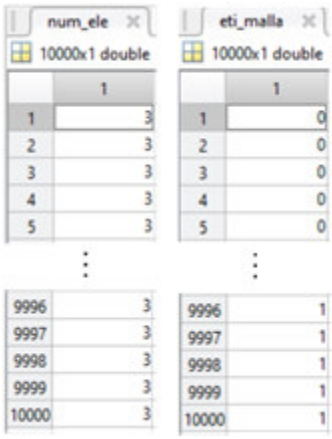


salida	
200x1 double	
	1
1	3
2	3
3	3
4	2
5	3
⋮	⋮
196	2
197	3
198	3
199	3
200	3

eti_pre_test	
200x1 double	
	1
1	0
2	0
3	0
4	0
5	0
⋮	⋮
196	1
197	1
198	1
199	1
200	1

Figura 2.137 Número de votos ganadores con su respectiva etiqueta predicha en muestras de pruebas.

```
% Creación de malla para observar la frontera de decisión del clasificador
minimos = min(XY_Test);
maximos = max(XY_Test);
mallax = linspace(minimos(1),maximos(1),100);
mallay = linspace(minimos(2),maximos(2),100);
coordenadas = [mallax.' mallay.'];
[u, v] = meshgrid(mallax, mallay);
malla = [u(:)'; v(:)'];
mallaz = malla.';
[num_ele, eti_malla] = knn(k,XY_Train,etiqueta_Train,mallaz);
```



num_ele	
10000x1 double	
	1
1	3
2	3
3	3
4	3
5	3
⋮	⋮
9996	3
9997	3
9998	3
9999	3
10000	3

eti_malla	
10000x1 double	
	1
1	0
2	0
3	0
4	0
5	0
⋮	⋮
9996	1
9997	1
9998	1
9999	1
10000	1

Figura 2.138 Número de votos del ganador con su etiqueta en conjunto que realiza la malla.

```
malla_aux = rescale(num_ele,0,0.8);
Et_malla = eti_malla + malla_aux;


Et_malla = reshape(Et_malla, size(u,1),size(u,2));
```

En Figura 2.139 se aprecia el resultado de la gráfica con las muestras etiquetas de acuerdo a la predicción del algoritmo

```
% Graficas de resultado de datos de prueba
Graficar(eti_pre_test, etiqueta_Test, XY_Test...
    ,Et_malla+length(unique(etiqueta_Test)), coordenadas)
```

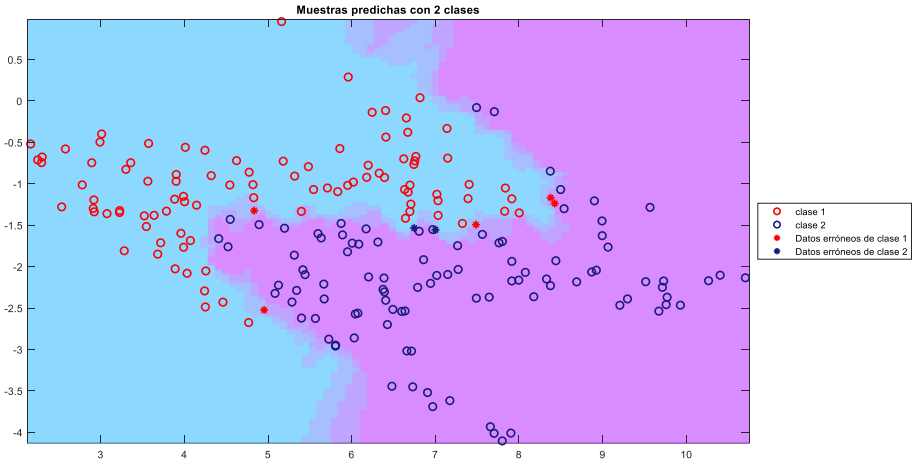


Figura 2.139 Muestras predichas por algoritmo kNN

Los resultados del Dataset de prueba se pueden apreciar en la Figura 2.140.

```
[~,Result]= confusion.getMatrix(etiqueta_Test,eti_pre_test);
```

```
Métricas:
Exactitud: 0.965000
Precisión: 0.979381
Sensibilidad: 0.950000
Especificidad: 0.980000
Medida F1: 0.964467
Porcentaje de aciertos para dataset de prueba es: 96.50%
```

Figura 2.140 Resultados del dataset de prueba

Tenemos en la Figura 2.141 a la matriz de confusión.

```
figure
cm = confusionchart(etiqueta_Test,eti_pre_test,'RowSummary','row-normalized',...
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
fprintf('Porcentaje de aciertos para dataset de prueba es: %.2f%% \n ',...
    Result.Accuracy*100);
```

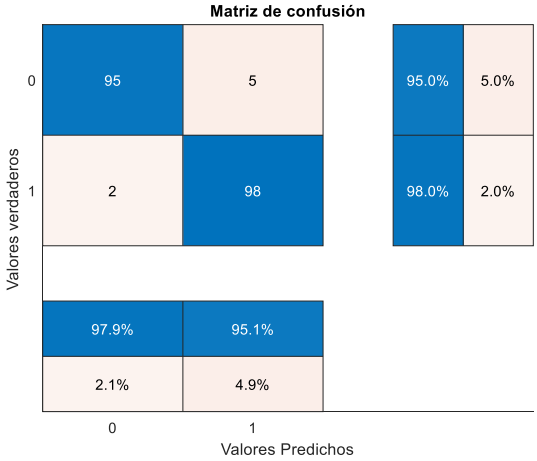


Figura 2.141 Matriz de confusión con datos predichos por algoritmo Naive Bayes

2.3.9. PROBABILISTIC NEURAL NETWORK

A continuación, se explica el código implementado en el algoritmo PNN cuya teoría se se encuentra disponible en la página 36, además se pretende que la arquitectura de la red se base en la Figura 1.32, este se dividirá en:

- Función “Graficar” (ya ha sido explicada)
- Función “Confusión” (ya ha sido explicada)
- Función “predicedistancia” (ya ha sido explicada)
- Función “PNN”, que es el programa principal del algoritmo.

```
%% Red probabilística
% Ingreso de Datos
clc, clear, close all;
```

Primero se cargan los datos que se van a usar.

```
load('Entrenamiento.mat');
```

Se carga un conjunto de datos el cual se supone que contendrán las etiquetas conocidas.

```
datos_entrenar = train2c500;
```

Se separan los datos de las coordenadas de las etiquetas.

```
XY_Train = datos_entrenar(:,1:2);
```

En un vector se coloca las etiquetas de cada muestra

```
etiqueta_Train = datos_entrenar(:,3)+1;
```

Se carga un conjunto de datos de pruebas de la misma manera que en el apartado anterior.

```
load('Prueba.mat');

datos_prueba = test2c;
XY_Test = datos_prueba(:,1:2);
```

```
etiqueta_Test = datos_prueba(:,3)+1;
```

Se inicializa el array clases que contiene las etiquetas respectivas de cada una, además el número de cada una.

```
clases = unique(etiqueta_Train);
nclases = length(clases);
```

Se inicializa el valor de sigma en 0.5 debido a que nos ha dado mejores resultados. Inicializamos los valores de las clases, tenemos el número de clases y la constante.

```
sigma = 0.5;
```

```
clases =    nclases =    sigma =

     1           2          0.5000
     2
```

En el vector "`mxclase`" se guardarán el número de muestras por cada clase.

```
mxclase=zeros(nclases,1);     % Vector contiene número de clases
```

Se busca el número de muestras por cada clase en el vector inicializado anteriormente.

```
for i=1:nclases          % en cada clasificación
    ind=find(etiqueta_Train==i);
    mxclase (i)=length(ind);
end
```

Se calcula la constante de la base. De la Ecuación (1.62).

```
c_bas = 1/(sqrt(2*pi)*sigma);
```

Se calcula la constante del denominador del exponente de la fórmula.

```
c_exp = 2*(sigma).^2;
```

```
mxclase =    c_bas =      c_exp =

   250          0.7979       0.5000
   250
```

Se inicializa vectores:

`Wdist`: contiene los pesos de la capa de suma.

`Eti_pre`: contiene las etiquetas predichas por el algoritmo

`Salida`: contiene la salida sin etiquetar de la red probabilística.

```
w_dist = zeros(1,nclases,1);
eti_pre = zeros(1,size(XY_Test,1));
salida = zeros(1,size(XY_Test,1));
```

Se comienza iterando cada muestra del dataset de entrada.

```
for muestra = 1: size(XY_Test,1)
```

Se itera entre las muestras de cada clase del conjunto con etiquetas conocidas.

```
for i=1:nclases
    auxClases = XY_Train(etiqueta_Train == clases(i),:);
    distancia = zeros(1,mxclase(i));
```

Para la capa de patrones a cada muestra se calcula la distancia entre cada muestra con etiqueta conocida y la muestra de prueba a la que se quiere clasificar como se observa en la Figura 2.142.

```
for cont  = 1: mxclase(i)
   distancia(cont) = sqrt(sum((XY_Test(muestra,:)-...
       auxClases(cont,:)).^2));
```

distancia — 1x250 double

	1	2	3	4	5	...	246	247	248	249	250
1	4.8454	4.3939	4.4568	4.4318	4.5601		0.8405	0.4886	0.5130	0.6464	0.5043

Figura 2.142 distancia de a cada punto de entrenamiento

```
    end
```

La capa se suma esta resumida en la siguiente línea de código. En la Figura 2.143 podemos ver el resultado de la capa de suma.

```
    w_dist(i) = (1/mxclase(i))*sum(c_bas * exp(-distancia/c_exp));
end
```

w_dist — 1x2 double

	1	2
1	0.0195	0.0023

Figura 2.143 Pesos de la capa de suma

La capa de salida esta resumida en la siguiente línea de código. En la Figura 2.144 podemos ver los valores de la capa de salida.

```
salida(muestra) = max(w_dist);
```

salida — 1x200 double

	1	2	3	4	5	...	196	197	198	199	200
1	0.0195	0.1460	0.1465	0.1444	0.1300		0.1035	0.1029	0.1416	0.1255	0.1536

Figura 2.144 Valores a la salida de la capa de salida

Finalmente, se etiqueta a cada muestra en base a la salida máxima de cada muestra de prueba. En la Figura 2.145 se muestra el vector de etiquetas correspondiente al dataset de pruebas.

```
eti_pre(muestra) = find(w_dist==max(w_dist);
```

eti_pre — 1x200 double

	1	2	3	4	5	...	196	197	198	199	200
1	1	1	1	1	1		2	2	2	2	2

Figura 2.145 Etiqueta a cada muestra en bases a la salida máxima de cada muestra de prueba.

```
end
```

A continuación, en la Figura 2.146 se grafica su respectiva matriz de confusión.

```
figure

cm = confusionchart(etiqueta_Test,eti_pre.','RowSummary','row-normalized',...
    'ColumnSummary','column-normalized');
cm.Title = 'Matriz de confusión';
cm.XLabel = 'Valores Predichos';
cm.YLabel = 'Valores verdaderos';
[~,Result]= confusion.getMatrix(etiqueta_Test, eti_pre);
fprintf('Porcentaje de aciertos para dataset de prueba es: %.2f%% \n ',...
    Result.Accuracy*100);
```

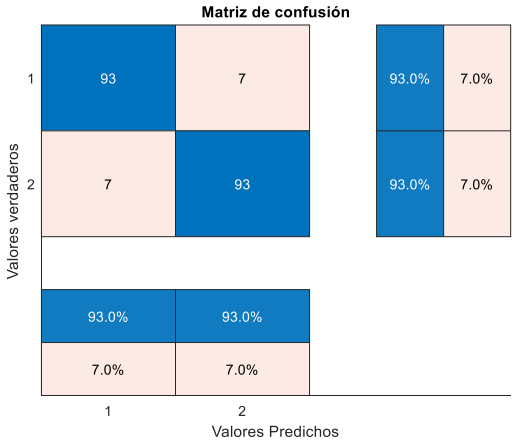


Figura 2.146 Matriz de confusión con datos predichos por algoritmo PNN

Para medir de alguna forma el nivel de sobreajuste del algoritmo se clasifica las muestras de entrenamiento asumiendo que no conocemos sus etiquetas. Y se siguen los pasos anteriormente mencionados para sacar sus etiquetas y salidas.

```
%Datos de entrenamiento
w_dist = zeros(1,nclases,1);
eti_Train = zeros(1,size(etiqueta_Train,1));
salida = zeros(1,size(etiqueta_Train,1));
for muestra = 1: size(etiqueta_Train,1)
    for i=1:nclases
        auxClases = XY_Train(etiqueta_Train == clases(i),:);
        distancia = zeros(1,mxclase(i));
        for cont  = 1: mxclase(i)
            distancia(cont) = sqrt(sum((XY_Train(muestra,:)-...
                auxClases(cont,:)).^2));
        end
        w_dist(i) = (1/mxclase(i))*sum(c_bas * exp(-distancia/c_exp));
    end
    salida(muestra) = max(w_dist);
    eti_Train(muestra) = find(w_dist==max(w_dist));
end

[~,Result]= confusion.getMatrix(etiqueta_Train, eti_Train);
fprintf('Porcentaje de aciertos para dataset de entrenamiento es: %.2f%% \n ',...
    Result.Accuracy*100);
```

```
Porcentaje de aciertos para dataset de entrenamiento es: 87.80%
```

Se crea la malla característica para una visualización más intuitiva, no se va a explicar el código porque ya se ha hecho en varios códigos atrás.

```
% Creación de malla para observar la frontera de decisión del clasificador
minimos = min(XY_Test);
maximos = max(XY_Test);
mallax = linspace(minimos(1),maximos(1),200);
mallay = linspace(minimos(2),maximos(2),200);
coordenadas = [mallax.' mallay.'];
[u, v] = meshgrid(mallax, mallay);
malla = [u(:)'; v(:)'];
mallaz = zscore(malla.');
```

Se clasifica cada punto de la malla del mismo modo que hicimos para el dato de prueba.

```
eti_malla = zeros(1,size(XY_Test,1));
sal_malla = zeros(1,size(XY_Test,1));
for muestra = 1: size(mallaz,1)
    for i=1:nclases
        auxClases = XY_Train(etiqueta_Train == clases(i),:);
        distancia = zeros(1,mxclase(i));
        for cont  = 1: mxclase(i)
            distancia(cont) = sqrt(sum((mallaz(muestra,:)-...
                auxClases(cont,:)).^2));
        end
        w_dist(i) = (1/mxclase(i))*sum(c_bas * exp(-distancia/c_exp));
    end
    sal_malla(muestra) = max(w_dist);
    eti_malla(muestra) = find(w_dist==max(w_dist));
end

mallaprese = rescale(sal_malla,0,1);
Ymalla = eti_malla + mallaprese;
Ymalla = reshape(Ymalla, size(u,1),size(u,2));
```

Finalmente, en la Figura 2.147 observamos como se grafica las muestras de prueba con sus errores y su respectiva malla.

```
Graficar(class-1, etiqueta_Test.'-1, XY_Test.',Ymalla-1+...
    length(unique(etiqueta_Test)), coordenadas)
```

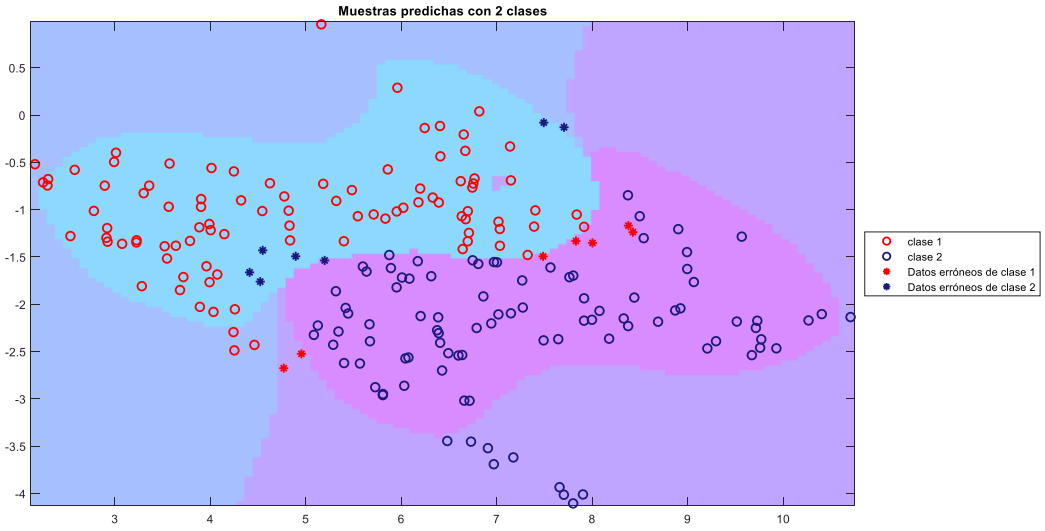


Figura 2.147 Resultados de clasificar datos con PNN

3. RESULTADOS Y DISCUSIÓN

3.1. COMPARACIÓN CUANDO EXISTEN 2 CLASES

A continuación, se presentarán los resultados más relevantes de probar los algoritmos de clasificación. Si se quiere reproducir los resultados puede correr los programas. Un manual de usuario se encuentra en el ANEXO A.

3.1.1. RESULTADOS EN FUNCIÓN DE NÚMERO DATOS DE ENTRENAMIENTO

Cuando se entrenan los algoritmos con datasets que tienen distintos números de muestras se tiene:

Tabla 3.1 Exactitud en función del número de datos de entrenamiento

Algoritmo	50 muestras	100 muestras	200 muestras	500 muestras	700 muestras
MLP	0,83	0,86	0,89	0,91	0,93
SVM	0,88	0,90	0,90	0,91	0,90
DT	0,91	0,94	0,95	0,96	0,97
NB	0,82	0,82	0,84	0,85	0,85
KNN	0,94	0,97	0,93	0,97	0,98
PNN	0,94	0,95	0,92	0,93	0,94

De la Tabla 3.1 se puede ver que los valores más altos que alcanza la exactitud en los 4 primeros algoritmos los alcanza cuando existe una gran cantidad de muestras de entrenamiento. Mientras que en KNN y PNN la exactitud tiene valores altos independientemente de la cantidad de muestras. Por lo tanto, cuando se tiene un dataset de entrenamiento grande el mejor algoritmo de clasificación es el Decision tree. Por otro lado, cuando se tiene menos muestras de entrenamiento, el mejor algoritmo a elegir es el KNN.

Tabla 3.2 Cantidad de aciertos de dataset de prueba cuando se entrenan datasets de distintos tamaños

Algoritmo	50 muestras	100 muestras	200 muestras	500 muestras	700 muestras
MLP	166	171	178	181	185
SVM	176	179	180	181	179
DT	182	187	189	191	193
NB	163	164	168	170	170
KNN	188	193	186	193	196
PNN	188	189	183	186	187

De la se puede ver que los valores más altos que alcanza la exactitud en los 4 primeros algoritmos los alcanza cuando existe una gran cantidad de muestras de entrenamiento. Mientras que en KNN y PNN la exactitud tiene valores altos independientemente de la cantidad de muestras. Por lo tanto, cuando se tiene un dataset de entrenamiento grande el mejor algoritmo de clasificación es el Decision tree. Por otro lado, cuando se tiene menos muestras de entrenamiento, el mejor algoritmo a elegir es el KNN.

Tabla 3.2 se puede ver que el algoritmo KNN tiene mayor cantidad de aciertos cuando se prueba el algoritmo con el dataset de prueba, sigue el algoritmo DT y finalmente el algoritmo PNN.

Tabla 3.3 Precisión en función del número de datos de entrenamiento

Algoritmo	50 muestras	100 muestras	200 muestras	500 muestras	700 muestras
MLP	0,837	0,838	0,875	0,901	0,929
SVM	0,865	0,883	0,917	0,901	0,899
DT	0,873	0,958	0,959	0,960	0,960
NB	0,806	0,808	0,854	0,872	0,872
KNN	0,932	0,989	0,891	0,979	0,990
PNN	0,940	0,968	0,911	0,930	0,939

En Tabla 3.3 al ver la precisión se puede observar que al igual que las demás tablas el algoritmo KNN tiene mejores resultados en general cuando probamos el dataset de prueba.

3.1.2. RESULTADOS EN FUNCIÓN DE LA DISTANCIA DE LOS DATOS DE PRUEBA

Tomando en cuenta la cantidad de aciertos por cada algoritmo.

Tabla 3.4 Número de aciertos en función de la distancia de los datos de prueba a la frontera de decisión

Algoritmo	Cerca	Medio	Lejos
MLP	116	195	199
SVM	115	189	194
DT	150	198	198
NB	105	180	200
KNN	166	200	200
PNN	130	196	200

En la Tabla 3.4 se puede ver que en todos los algoritmos tienden a clasificar mejor a medida que las muestras de prueba se alejan de la frontera de decisión. El algoritmo que tiene un mayor número de aciertos cuando las muestras están cercanas a la frontera de decisión es el KNN. Cuando las muestras se encuentran a media distancia y lejanas de la frontera se puede apreciar que el KNN también tiene mayor número de aciertos.

¿Qué pasa con el valor de las salidas?

A continuación, una tabla con el promedio de valores de salida, nos referimos a estos como los valores que se nos arrojan los algoritmos antes de que estos sean etiquetados, por ejemplo, en el caso de la MLP nos saldrán valores de 0 al 1 que indican la probabilidad de que los datos pertenezca a una clase o a otra, se etiquetara de acuerdo a la salida que tenga la mayor probabilidad, de estas se hace el promedio como en la Tabla 3.5, algo similar pasa en el algoritmo SVM pero no arrojan probabilidades si no la supuesta distancia a la que se encuentra en la muestra de la frontera de decisión del algoritmo. En el caso de DT se puede notar que el algoritmo directamente clasifica las muestras, no indica una aproximación. En el caso del KNN, se puede observar que obtenemos el promedio de la cantidad de votos con los que ganó una clase al clasificarla.

Tabla 3.5 Promedio de datos de salida en función de la distancia de los datos de prueba a la frontera de decisión

Algoritmo	Cerca	Medio	Lejos
MLP	0,80	0,84	0,92
SVM	0,91	1,37	1,71
DT	-	-	-
NB	0,0353	0,0353	0,0175
KNN	2,47	3,00	3,00
PNN	0,058	0,071	0,033

Ya que se tienen distintas escalas no se puede hacer una comparación, por lo tanto, se realiza una normalización respecto con el valor máximo de cada fila.

De la Tabla 3.5 se puede concluir que el algoritmo que proporciona mejor información extra sobre el alejamiento a la frontera de decisión es el SVM, debido a que sus valores se ven claramente marcados entre cada conjunto de datos, mientras que el árbol de decisión no nos muestra ningún dato porque su salida es directamente la etiqueta de cada muestra.

En la Tabla 3.6 podemos ver al normalizar los valores de salida promedio y al restar los valores a una distancia cercana de una media se puede ver que en el algoritmo SVM tiene una diferencia más grande, mientras que en la diferencia entre datos que están lejos y que están a una distancia media se puede ver que en SVM hay una diferencia considerable, en la MLP hay una pequeña diferencia mientras que en NB, KNN, PNN, la diferencia es cero incluso negativa, por lo que una hay un patrón creciente que muestra claramente que se está alejando de la frontera de decisión.

Tabla 3.6. Valores Normalizados del promedio de las salidas

Algoritmo	Cerca	Medio	Lejos	Medio - Cerca	Lejos -Medio
MLP	0,87	0,91	1,00	0,04	0,09
SVM	0,54	0,80	1,00	0,26	0,20
DT	-	-	-	-	-
NB	1	1	0,050	0,00	-0,95
KNN	0,82	1,00	1,00	0,18	0,00
PNN	0,815	1,000	0,459	0,19	-0,54

3.1.3. RESULTADOS EN FUNCIÓN DE LA CANTIDAD DE DATOS ASIMÉTRICOS DE ENTRENAMIENTO

Primero se verá qué pasa con la exactitud cuando probamos a cada algoritmo con el dataset de prueba.

Tabla 3.7 Exactitud en función de la cantidad de datos asimétricos

Dataset entrenamiento	Relación de clases 1:2	Relación de clases 1:3	Relación de clases 1:4
MLP	0,88	0,79	0,75
SVM	0,85	0,80	0,75
DT	0,97	0,94	0,96
NB	0,84	0,85	0,83
KNN	0,94	0,94	0,97
PNN	0,92	0,92	0,92

En la Tabla 3.7 se puede ver que a medida que la relación entre clases crece la exactitud para el dataset de pruebas disminuye en los algoritmos, excepto por 3 estos son NB, DT, KNN, y PNN, en estos tres los valores se mantienen y no decaen. Por lo tanto, si se quiere que la exactitud se quede en valores elevados a pesar de la cantidad de muestras asimétricas los 3 últimos algoritmos nombrados son una buena opción.

Tabla 3.8 Precisión en función de la cantidad de datos asimétricos

Dataset entrenamiento	Relación de clases 1:2	Relación de clases 1:3	Relación de clases 1:4
MLP	0,99	0,99	1,00
SVM	1,00	1,00	1,00
DT	0,94	0,94	0,96
NB	0,93	0,97	0,97
KNN	0,97	0,97	1,00
PNN	0,93	0,95	0,93

En la Tabla 3.8, como era de esperarse la precisión es muy alta, pero hay un algoritmo que gana y es el SVM donde no se equivoca en ninguna muestra cuando en la clase 2 hay más muestras que en la clase 1.

Para la Sensibilidad se tiene:

Tabla 3.9 Sensibilidad en función de la cantidad de datos asimétricos

Algoritmo	Relación de clases 1:2	Relación de clases 1:3	Relación de clases 1:4
MLP	0,76	0,58	0,50
SVM	0,69	0,60	0,50
DT	0,99	0,94	0,96
NB	0,75	0,72	0,68
KNN	0,91	0,91	0,94
PNN	0,90	0,89	0,90

En la Tabla 3.9, como se puede ver en la tabla para la Sensibilidad en varios algoritmos es baja, excepto en los algoritmos DT, KNN, PNN, se puede observar que funcionan mejor que los demás algoritmos a pesar de que se tienen menos muestras que la clase 1.

Tabla 3.10 Medida F1 en función de la cantidad de datos asimétricos

Dataset entrenamiento	Relación de clases 1:2	Relación de clases 1:3	Relación de clases 1:4
Número de muestras de prueba	200 muestras	200 muestras	200 muestras
MLP	0,86	0,73	0,67
SVM	0,82	0,75	0,67
DT	0,97	0,94	0,94
NB	0,82	0,82	0,80
KNN	0,92	0,94	0,97
PNN	0,91	0,92	0,91

En la Tabla 3.10 se puede ver que la medida de relación entre la especificidad y la sensibilidad es la medida f1, por lo tanto, cuando se tiene DT, KNN y PNN a medida que la relación no baja su valor en la medida f1 por lo tanto si se quiere algoritmos que no dependan del número de muestras de cada una de sus clases entonces la mejor opción es el KNN. Como se puede observar en la Tabla 3.4, la medida f1 tiene valores similares a la exactitud, por lo tanto, en este caso es mejor considerar la exactitud.

3.2. COMPARACIÓN CUANDO EXISTEN 4 CLASES

3.2.1. RESULTADOS EN FUNCIÓN DEL NÚMERO DE DATOS DE ENTRENAMIENTO

Ahora se evalúa la exactitud de cada algoritmo.

Tabla 3.11 Exactitud en función del número de datos de entrenamiento con 4 clases

Algoritmo	100 muestras	200 muestras	500 muestras	700 muestras
MLP	0,90	0,93	0,95	0,96
SVM	0,87	0,94	0,95	0,96
DT	0,88	0,89	0,91	0,91
NB	0,89	0,92	0,92	0,93
KNN	0,90	0,90	0,97	0,97
PNN	0,91	0,93	0,97	0,98

En la Tabla 3.11 se puede ver que en este caso el ganador el algoritmo PNN, en cuanto a la exactitud supera en a los demás algoritmos excepto cuando se tiene 200 muestras, en

ese caso gana el SVM, por lo tanto, cuando se tiene varias clases el algoritmo PNN es la mejor opción para clasificar, cuando se tiene una frontera de decisión.

También se tiene una tabla comparativa de todos los algoritmos:

Tabla 3.12 Comparación de algoritmos

Método	Multilayer Perceptron	Support Vector Machine	Decision Tree	Naive Bayes	k-Nearest Neighbors	Probabilistic Neural Network
Dificultad para implementar algoritmo desde cero	Alta	Alta	Media	Fácil	Fácil	Fácil
Problemas	- Si la red se entrena mal, sus salidas pueden llegar a ser imprecisas. -No se necesita escalado de datos - La función de error puede tener mínimos locales, entonces la red puede dejar de entrenarse sin que haya caído en un mínimo global.	- El algoritmo en principio solo puede clasificar en 2 clases. -No se necesita escalado de datos	- Propenso al sobreajuste. - Si hay algún dato diferente al hacer el árbol puede cambiar en gran medida el resultado.	-Problema de Frecuencia Cero - En la vida real no se puede tener un conjunto de muestras conocidas que sean completamente independientes	- Fácil de entender. - Alto nivel de precisión.	- Para nuevos casos son más lentas que las MLP. - Requerimos más espacio de memoria para almacenar el modelo.
Ventajas	- Puede trabajar bien con grandes cantidades de datos. - Se puede aplicar en problemas complejos	- Son ideales para problemas con muchas dimensiones.	-No se necesita escalado de datos necesarios. -No se necesita escalado de datos -Es un algoritmo intuitivo.	- Bueno con entradas categóricas. -No se necesita escalado de datos - Es fácil de implementar.	- Si existen gran cantidad de datos se volverá complicado usar este algoritmo. -No se necesita escalado de datos	- Son mucho más rápidas que las MLP. - Son insensibles a valores atípicos. - Llegan a ser más precisas que las MLP. -No se necesita escalado de datos

En la Tabla 3.12 se puede ver que los algoritmos más simples como los KNN o PNN a pesar de su simplicidad son muy efectivos para hacer modelos de Machine Learning

3.3. DISCUSIÓN

En [3]se puede observar el algoritmo con mayor exactitud al comparar entre 3 algoritmos de clasificación (SVM, KNN y NB) mostraron que el SVM tiene mejor resultados en sus métricas como sensibilidad, exactitud y especificidad. Se debe tomar en cuenta que cada muestra constaba de 26 características, o una muestra con 26 dimensiones. Por lo tanto, posiblemente si cada algoritmo tenga más características es probable que el SVM sea más útil que el KNN, ya que en este caso este último salió ganador en este trabajo.

En [41] se trata de averiguar cuál es el mejor algoritmo de clasificación para detectar enfermedades del corazón y hepatitis en donde se comparan LR, DT, NB, KNN, SVM y RF.

El resultado es favorable para Random Forest seguido por SVM, se debe aclarar que en este caso también se tienen varios atributos para el caso de enfermedades del corazón se tiene dataset de 14 atributos y 303 muestras, mientras para hepatitis se tiene 20 atributos y 155 muestras. Mientras que en este caso se está trabajando con 2 atributos y entre 50 y 100 muestras.

En [2] se comparan 3 algoritmos para un dataset de 220 muestras y 20 líneas, que vienen a representar las características de que cada muestra, en este caso entre el KNN, NB y SVM se tiene que el que da mayor exactitud a la hora de clasificar es el NB, a diferencia de este caso que presenta los valores más bajos.

En [7] se trata de determinar personas que tienen diabetes según 9 atributos, se probarán 2 datasets uno con 384 muestras y otro con 768, aquí se enfrentaron SVM, NB, MLP, RF, JRip, y DT. En este caso ganó SVM para los datasets a probar seguido de RF y MLP que fueron los mejores para esos mientras que en este caso se tiene a KNN como ganador debido a que se tiene solo 2 atributos para diferentes conjuntos de datos.

4. CONCLUSIONES Y RECOMENDACIONES

4.1. CONCLUSIONES

DETERMINACIÓN DEL GRADO DE PERTENENCIA

Se investigó si cada unos de los algoritmos permiten conocer el grado de pertenencia a la clase seleccionada o solamente dice la clase a la que pertenece.

Al momento de evaluar los algoritmos en función de la distancia de los datos de prueba se puede observar en todos los algoritmos que a menor distancia de la frontera de decisión la la cantidad de aciertos disminuye. En la Tabla 3.4 el algoritmo KNN se realza sobre los demás con 166 aciertos cuando las muestras de prueba están cerca de la frontera de decisión.

Cuando las muestras de prueba están a una distancia media de la frontera de decisión se puede ver un incremento en el número de aciertos en todos los algoritmos, pero en el KNN se puede apreciar que tiene los valores más altos llegando a tener 200 aciertos de 200 muestras de prueba.

Cuando las muestras están lejos de la frontera de decisión en todos los algoritmos presentan un gran número de aciertos, pero el NB, KNN y PNN presentan 200 aciertos de 200 muestras.

Al momento de evaluar la salida de los algoritmos cuando los algoritmos, en la Tabla 3.5 se puede ver que la salida de la MLP y SVM tiene una tendencia clara de aumentar a medida que se alejan de la frontera de decisión es decir que mientras más se alejan los datos a ser probados, el algoritmo está más seguro de que clasifica correctamente. Un ejemplo de ello se puede ver en el SVM donde la salida del algoritmo tiene 0.91 de salida cuando está cercano a la frontera de decisión, 1.37 a una distancia media y 1.71 cuando está lejos de la frontera de decisión.

CANTIDAD DE LAS MUESTRAS EN EL DATASET DE ENTRENAMIENTO Y SU IMPLICACIÓN AL MOMENTO DE PROBAR EL ALGORITMO CON DATASET DE PRUEBA

Se investigó si la cantidad de datos de ENTRENAMIENTO influye en la exactitud de los métodos. Inicialmente se planteó que no es lo mismo que se disponga de cientos de datos a que solo se tengan decenas de estos.

En los algoritmos mencionados anteriormente se puede observar que mientras más muestras de entrenamiento se tenga, algunos algoritmos tienden a aumentar su exactitud.

Como se puede apreciar en la Tabla 3.1, de entre todos los algoritmos se destaca el KNN que presenta una exactitud del 94% con 50 muestras de entrenamiento, y cuando se se tienen 700 muestras se llega al 98%.

Se puede observar con 4 clases al momento de entrenar según el número de muestras de entrenamiento, que en todos los algoritmos hay una tendencia de la exactitud a aumentar a medida que se incrementa el número de muestras de entrenamiento. En la Tabla 3.11 se puede apreciar valores altos en la exactitud en todos los casos independientemente del algoritmo, pero el que resalta sobre los demás es el PNN que llega al 91% cuando se tiene 100 muestras de entrenamiento, 93% cuando se tienen 200 muestras, 97% para 500 muestras y 98% cuando se tienen 700 muestras.

ASIMETRÍA EN LOS CONJUNTOS DE ENTRENAMIENTO

Se investigó si la cantidad de datos de entrenamiento influye en la exactitud de los métodos. Inicialmente se planteó que no es lo mismo que se disponga de conjuntos de datos de entrenamiento que contienen el mismo número de muestras para cada una de sus clases que en otros casos en los que la diferencia entre clases es considerable.

Cuando se tiene una cantidad de datos asimétrica a la hora de entrenar el algoritmo se puede ver en la Tabla 3.7 que en algunos algoritmos que la exactitud disminuye a medida que la relación entre clase aumenta, un ejemplo de ello se encuentra en los algoritmos MLP, SVM y NB. En el caso de la MLP se tiene que la exactitud con una relación entre clases de 1:2 es del 88% mientras que cuando en una relación de 1:4 esta disminuye a 0.75. En algoritmos como DT, KNN y PNN se mantienen constantes valores e incluso llegan a aumentar, como es el caso del KNN que en una relación de 1:2 llega del 94% mientras que en una relación se tiene un 97%.

Para la precisión en todos los algoritmos se mantiene en un valor alto, en la Tabla 3.8 se puede observar que el más alto se obtiene con la SVM que está llegando al 100% en todos los casos.

En la sensibilidad se puede percibir valores bajos excepto en DT, KNN, y PNN, que mantienen sus valores altos a pesar de que tienen pocas muestras para entrenar, el caso más alto se dá en el Decision Tree llegando al 99% cuando la relación entre clases es de 1:2 mientras que en 1:3 llega a 94% y 1:4 tiene un 96%.

4.2. RECOMENDACIONES

Como se han visto en estudios anteriores tendríamos que probar algoritmos de clasificación supervisada cuando las muestras tienen varias dimensiones o atributos, posiblemente sea

un factor que influya decisivamente al momento de comprar algoritmos y los que posiblemente funcionan bien cuando se tiene solo 2 dimensiones sean menos efectivos cuando variamos el número de estas.

En un futuro se puede evaluar los métodos de cada algoritmo, por ejemplo, el algoritmo SVM se llega a una ecuación que se puede resolver de maneras diferentes, tomando otro ejemplo sería el Decision Tree, en este algoritmo se puede utilizar distintas técnicas para construir el árbol, se podrían comparar esas técnicas para determinar cuál da mejores resultados.

5. REFERENCIAS BIBLIOGRÁFICAS

[1] Christian Guardiola González, Patricio Letelier Torres, and Mercedes García Martínez, “Clasificador de texto mediante técnicas de aprendizaje automático,” Universitat Politècnica de València, Valencia, 2020.

[2] A. M. Ibrahim, M. Ezzat, and A. Y. Abdelaziz, “Performance comparison of classification methods for line outage detection,” in *2016 Eighteenth International Middle East Power Systems Conference (MEPCON)*, 2016, pp. 26–32, doi: 10.1109/MEPCON.2016.7836867.

[3] A. Bourouhou, A. Jilbab, C. Nacir, and A. Hammouch, “Comparison of classification methods to detect the Parkinson disease,” in *2016 International Conference on Electrical and Information Technologies (ICEIT)*, 2016, pp. 421–424, doi: 10.1109/EITech.2016.7519634.

[4] P. Perera, Y.-C. Tian, C. Fidge, and W. Kelly, “A Comparison of Supervised Machine Learning Algorithms for Classification of Communications Network Traffic,” in *Neural Information Processing*, 2017, pp. 445–454.

[5] H. Bouali and J. Akaichi, “Comparative Study of Different Classification Techniques: Heart Disease Use Case,” in *2014 13th International Conference on Machine Learning and Applications*, 2014, pp. 482–486, doi: 10.1109/ICMLA.2014.84.

[6] P. C. P. Júnior, A. Monteiro, R. D. L. Ribeiro, A. C. Sobieranski, and A. Von Wangenheim, “Comparison of Supervised Classifiers and Image Features for Crop Rows Segmentation on Aerial Images,” *Appl. Artif. Intell.*, vol. 34, no. 4, pp. 271–291, 2020, doi: 10.1080/08839514.2020.1720131.

[7] J. E. T. Akinsola, “Supervised Machine Learning Algorithms: Classification and Comparison,” *Int. J. Comput. Trends Technol.*, vol. 48, pp. 128–138, 2017, doi: 10.14445/22312803/IJCTT-V48P126.

[8] “Redes Neuronales: una visión superficial - Fernando Sancho Caparrini.” http://www.cs.us.es/~fsancho/?e=72 (accessed Jan. 19, 2022).

[9] L. H. Tsoukalas and R. E. Uhrig, *Fuzzy and neural approaches in engineering, matlab supplement*, 1st ed. [S.l.] : Wiley-Interscience,.

[10] Roque Molina Lega, “FÓRMULA DE TAYLOR EN FUNCIONES DE VARIAS VARIABLES.”

[11] “¿Qué es el Z-Score y como lo puedo aplicar a mi sistema de trading?”

https://traderprofesional.com/que-es/z-score/ (accessed Jan. 30, 2022).

[12] Baccam Nina, “Evitar el sobreajuste y los datos desequilibrados con AutoML,” Oct. 06, 2021. https://docs.microsoft.com/es-es/azure/machine-learning/concept-manage-ml-pitfalls (accessed Oct. 25, 2021).

[13] H. Li, Y. Liang, and Q. Xu, “Support vector machines and its applications in chemistry,” *Chemom. Intell. Lab. Syst.*, vol. 95, no. 2, pp. 188–198, 2009, doi: https://doi.org/10.1016/j.chemolab.2008.10.007.

[14] G. Aksoy and M. Karabatak, “Performance Comparison of New Fast Weighted Naïve Bayes Classifier with Other Bayes Classifiers,” in *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, 2019, pp. 1–5, doi: 10.1109/ISDFS.2019.8757558.

[15] Y. Sun, “CS145: INTRODUCTION TO DATA MINING,” Jan. 2019, Accessed: Oct. 26, 2021. [Online]. Available: http://web.cs.ucla.edu/~yzsun/classes/2019Wi.

[16] Baeldung, “Multiclass Classification Using Support Vector Machines | Baeldung on Computer Science,” Dec. 09, 2019. https://www.baeldung.com/cs/svm-multiclass-classification (accessed Apr. 13, 2021).

[17] K.-W. Chang, “Multi-Class Classification,” 2017, Accessed: Jan. 30, 2022. [Online]. Available: https://uclanlp.github.io/CS269-17/slides/CS269-03.pdf.

[18] Chang Kai-Wei, “Lecture 3: Multi-Class Classificat.” https://uclanlp.github.io/CS269-17/slides/CS269-03.pdf (accessed Apr. 13, 2021).

[19] C.-J. Hsieh, “ECS289: Scalable Machine Learning,” Oct. 2015.

[20] P. Dangeti, *Statistics for Machine Learning: Techniques for Exploring Supervised, Unsupervised, and Reinforcement Learning Models with Python and R*. Packt Publishing, 2017.

[21] “Machine Learning Decision Tree Classification Algorithm - Javatpoint.” https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm (accessed May 29, 2021).

[22] N. S. Chauhan, “Decision Tree Algorithm, Explained.” https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html (accessed May 29, 2021).

[23] S. K. K. Venkata and P. Kiruthika, “IJARCCE An Overview of Classification Algorithm in Data mining,” *Int. J. Adv. Res. Comput. Commun. Eng.*, vol. 4, pp.

255–257, Dec. 2015, doi: 10.17148/IJARCCE.2015.41259.

[24] R. Bhiksha, “Decision Trees,” 2016. https://www.cs.cmu.edu/~bhiksha/courses/10-601/decisiontrees/ (accessed Jan. 30, 2022).

[25] A. Mccallum and K. Nigam, “A Comparison of Event Models for Naive Bayes Text Classification,” Pittsburgh, 1998. Accessed: Jun. 20, 2021. [Online]. Available: https://www.cs.cmu.edu/~knigam/papers/multinomial-aaaiws98.pdf.

[26] A. Mucherino, P. J. Papajorgji, and P. M. Pardalos, “k-Nearest Neighbor Classification,” 1st ed., New York: Springer, 2009, pp. 83–106.

[27] “K Vecino más cercano.” https://datapeaker.com/big-data/k-vecino-mas-cercano-algoritmo-knn/ (accessed Jan. 30, 2022).

[28] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE Trans. Inf. Theory*, vol. 13, no. 1, pp. 21–27, 1967, doi: 10.1109/TIT.1967.1053964.

[29] X. Wu *et al.*, “Top 10 algorithms in data mining,” *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, 2008, doi: 10.1007/s10115-007-0114-2.

[30] S. Thirumuruganathan, “A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm,” 2010. https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/ (accessed Sep. 05, 2021).

[31] M. Alweshah and S. Abdullah, “Hybridizing firefly algorithms with a probabilistic neural network for solving classification problems,” *Appl. Soft Comput.*, vol. 35, pp. 513–524, 2015, doi: https://doi.org/10.1016/j.asoc.2015.06.018.

[32] B. Mohebali, A. Tahmassebi, A. Meyer-Baese, and A. H. Gandomi, “Chapter 14 - Probabilistic neural networks: a brief overview of theory, implementation, and application,” in *Handbook of Probabilistic Models*, P. Samui, D. Tien Bui, S. Chakraborty, and R. C. Deo, Eds. Butterworth-Heinemann, 2020, pp. 347–367.

[33] A. Norén, “5 maneras de lidiar con la falta de datos en el aprendizaje automático,” 2019. https://sitiobigdata.com/2019/12/24/5-maneras-de-lidiar-con-la-falta-de-datos-en-el-aprendizaje-automatico/# (accessed Oct. 25, 2021).

[34] “Matriz de confusión.” https://es.wikipedia.org/wiki/Matriz_de_confusión (accessed Oct. 25, 2021).

[35] “Matriz de confusión .” https://es.wikipedia.org/wiki/Matriz_de_confusión (accessed Jan. 31, 2022).

[36] Blanco Enrique, “¿Qué es overfitting y cómo evitarlo?,” Apr. 09, 2019. https://empresas.blogthinkbig.com/que-es-overfitting-y-como-evitarlo-html-2/ (accessed Oct. 25, 2021).

[37] J. I. Bagnato, “Qué es overfitting y underfitting y cómo solucionarlo.” https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/ (accessed Oct. 25, 2021).

[38] “Qué es overfitting y underfitting y cómo solucionarlo.” https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/ (accessed Jan. 31, 2022).

[39] “Overfitting - Wikipedia.” https://en.wikipedia.org/wiki/Overfitting (accessed Jan. 31, 2022).

[40] M. S. Abbas, “Multi Class Confusion Matrix,” Jul. 21, 2017. https://ww2.mathworks.cn/matlabcentral/fileexchange/60900-multi-class-confusion-matrix?s_tid=prof_contriblnk (accessed Dec. 14, 2021).

[41] C. A. Ul Hassan, M. S. Khan, and M. A. Shah, “Comparison of Machine Learning Algorithms in Data classification,” in *2018 24th International Conference on Automation and Computing (ICAC)*, 2018, pp. 1–6, doi: 10.23919/IConAC.2018.8748995.

ANEXOS

ANEXO A: MANUAL DE USUARIO

Para que sea más sencillo manejar los programas se ha creado un manual. Como se observa en la Figura A. 1 Carpetas de todos los programas, se tienen 7 carpetas para cada código, entre ellas tenemos:

- Creación de datos: que contiene los códigos para crear los datasets de prueba y dataset de entrenamiento.
- MLP: contienen archivos para probar el algoritmo Multilayer Perceptron.
- SVM: contienen los archivos para probar el algoritmo Support Vector Machine.
- DT: contienen los archivos para probar el algoritmo Decisión Tree.
- NB: contienen los archivos para probar el algoritmo Naive Bayes.
- KNN: contienen los archivos para probar el algoritmo K-Nearest Neighbor.
- PNN: contienen los archivos para probar el algoritmo Probabilistic Neural Network.

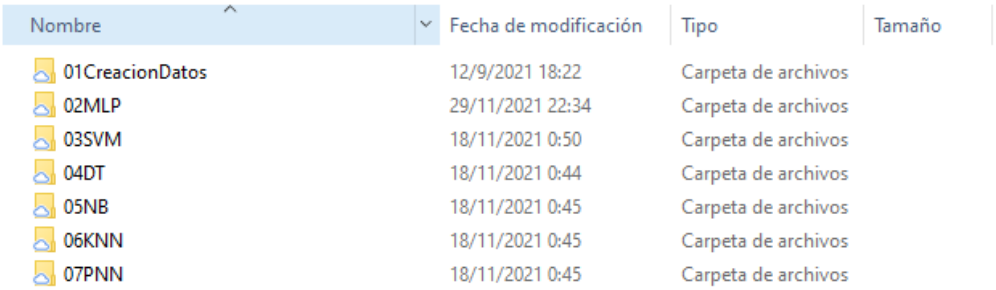


Figura A. 1 Carpetas de todos los programas

CREACIÓN DE DATOS

Como se observa en la Figura A. 2 cuando se abre la carpeta “01CreacionDatos”, se tienen 5 archivos. Entre ellos están:

- CreacionDatos.m: Es el programa principal en el cual se llaman a las funciones que generan datasets con 2 y 4 clases.
- Generacion2clases.m: es la función que genera datasets de 2 clases.
- Generacion4clases.m: es la función que genera datasets de 2 clases.
- Fronteras.mlx: es un pequeño programa con el que se puede graficar fronteras de decisión.
- GraficaDataset.mlx: es un programa con el que se puede graficar los datasets creados anteriormente.

Nombre	Fecha de modificación	Tipo	Tamaño
CreacionDatos.m	14/4/2021 12:14	MATLAB Code	3 KB
Fronteras.mlx	14/12/2021 11:34	MATLAB Live Script	65 KB
generacion2clases.m	14/12/2021 12:14	MATLAB Code	2 KB
generacion4clases.m	14/4/2021 12:25	MATLAB Code	4 KB
GraficaDataset.mlx	14/12/2021 12:13	MATLAB Live Script	67 KB

Figura A. 2 Archivos para la creación de datos

Para crear los datos de prueba y de entrenamiento se corre el programa dando clic en el ícono “Run”.(Figura A. 3)

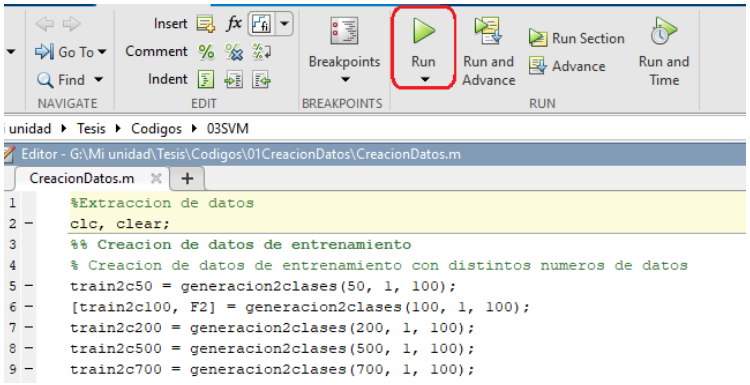


Figura A. 3 Archivo principal de creación de datos

Como se observa en la Figura A. 4 se crean 2 archivos, Entrenamiento.mat y Prueba.mat, que serán utilizados por los algoritmos. En este caso para evitar que se escriban diferentes direcciones al momento de abrir los datasets se copian los archivos en cada carpeta.

CreacionDatos.m	14/4/2021 12:14	MATLAB Code	3 KB
generacion4clases.m	14/4/2021 12:25	MATLAB Code	4 KB
Fronteras.mlx	14/12/2021 11:34	MATLAB Live Script	65 KB
GraficaDataset.mlx	14/12/2021 12:13	MATLAB Live Script	67 KB
generacion2clases.m	14/12/2021 12:14	MATLAB Code	2 KB
Entrenamiento.mat	14/12/2021 12:18	MATLAB Data	87 KB
Prueba.mat	14/12/2021 12:18	MATLAB Data	234 KB

Figura A. 4 Carpeta de creación de datos con archivos de datasets creados

Si se quiere graficar algún dataset sea de entrenamiento o de prueba, en la siguiente línea de código se puede cambiar el nombre del archivo, que está subrayado en amarillo.

```
load('Entrenamiento.mat');
```

En la siguiente línea de código se puede cambiar el nombre del dataset, que está subrayado en amarillo de acuerdo con la Tabla 1.1 y Tabla 1.2.

```
data = train2c200;
```

En la Figura A. 5 se puede ver el código completo, se puede apreciar que solo las dos líneas son necesarias para modificar el archivo. Y en la Figura A. 6 se presenta el resultado.

Live Editor - G:\Mi unidad\Tesis\Codigos\01CreacionDatos\GraficaDataset.mlx

CreacionDatos.m | GraficaDataset.mlx

```
load('Entrenamiento.mat');
data = train2c200;
figure
hold on
ele = 12;
colormap([1 0 .5; % magenta
          1 .5 0; % anaranjado
          0 .6 0; % verde
          0 0 .8]); % azul
datosclase1 = scatter(data(data(:,3)==0,1), data(data(:,3)==0,2),ele, data(data(:,3)==0,3));
datosclase2 = scatter(data(data(:,3)==1,1), data(data(:,3)==1,2),ele, data(data(:,3)==1,3));
title('Dataset con Frontera de decision rotada')
legend([datosclase1,datosclase2],{'clase 1','clase 2'})
hold off
```

Figura A. 5 Programa para graficar Datasets

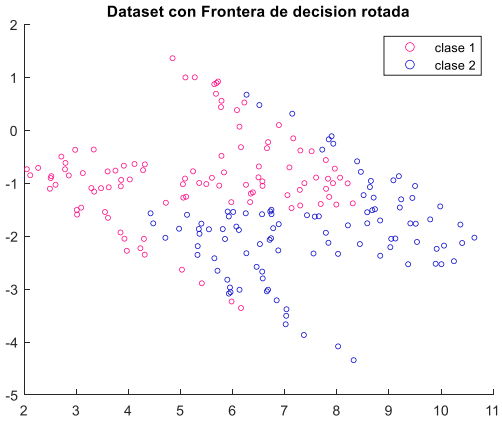
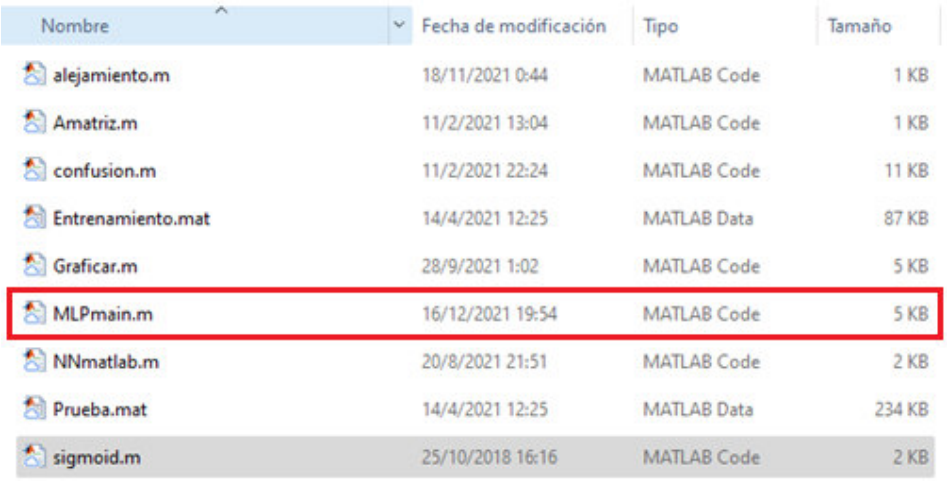


Figura A. 6 Dataset graficado

MLP

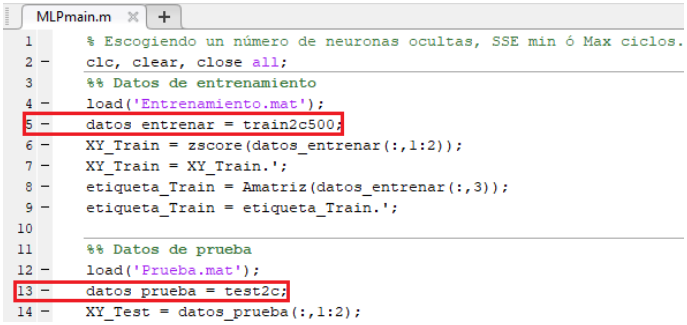
Para el algoritmo MLP se abre la carpeta 02MLP y se abre el archivo MLPmain.m donde está el programa principal. (Figura A. 7)



Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
Amatriz.m	11/2/2021 13:04	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	28/9/2021 1:02	MATLAB Code	5 KB
MLPmain.m	16/12/2021 19:54	MATLAB Code	5 KB
NNmatlab.m	20/8/2021 21:51	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB
sigmoid.m	25/10/2018 16:16	MATLAB Code	2 KB

Figura A. 7 Carpeta que almacena programa MLP

Si se quiere escoger un dataset de entrenamiento diferente se cambia el nombre del dataset de acuerdo a las Tablas Tabla 1.1Tabla 1.2, como se puede observar en la Figura A. 8 en la línea 5 y si se quiere cambiar el dataset de prueba se cambia la línea 13 del código por un nombre valido de la Tabla 1.3. Se debe recalcar que los datasets de entrenemiento y de prueba deben tener el mismo número de clases.



MLPmain.m

```
1    % Escogiendo un número de neuronas ocultas, SSE min ó Max ciclos.
2 -  clc, clear, close all;
3    %% Datos de entrenamiento
4 -  load('Entrenamiento.mat');
5 -  datos_entrenar = train2c500;
6 -  XY_Train = zscore(datos_entrenar(:,1:2));
7 -  XY_Train = XY_Train.';
8 -  etiqueta_Train = Amatriz(datos_entrenar(:,3));
9 -  etiqueta_Train = etiqueta_Train.';
10
11   %% Datos de prueba
12 -  load('Prueba.mat');
13 -  datos_prueba = test2c;
14 -  XY_Test = datos_prueba(:,1:2);
```

Figura A. 8 Líneas que se deben cambiar en programa MLP

Cuando se tienen los datasets correctos se corre el programa y se obtiene la Figura A. 9.

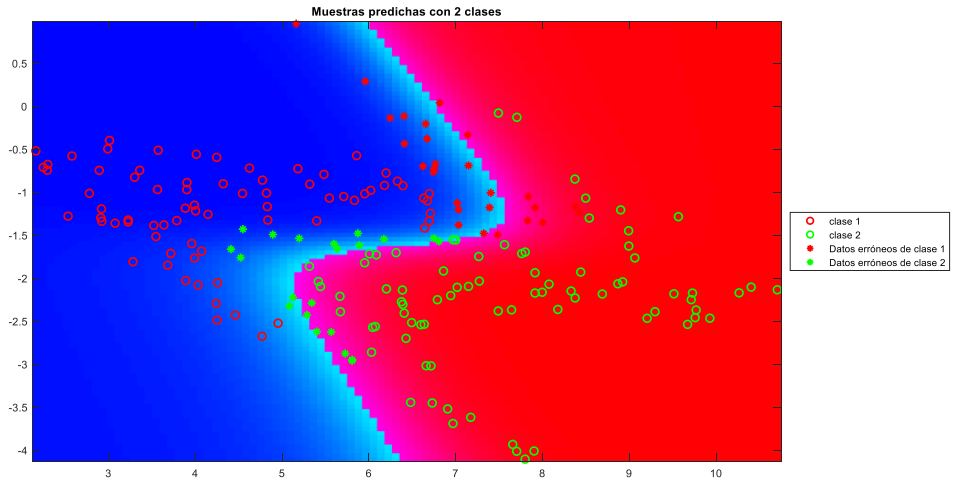


Figura A. 9 Gráfica resultante de correr programa MLP

En la Figura A. 10 se puede ver la matriz de confusión resultante y en Figura A. 11 se observa resultante de salidas de algoritmo.

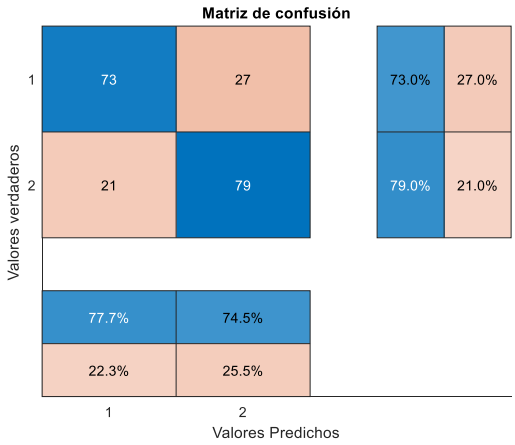


Figura A. 10 Matriz de confusión de MLP

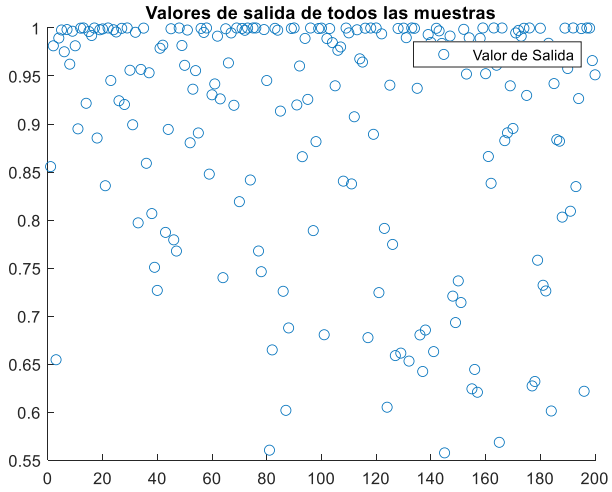


Figura A. 11 Valores de salidas de MLP

Si se quiere comparar con un algoritmo de las herramientas de Matlab se puede elegir el archivo NNmatlab y se puede correr de la misma manera que se hizo para el archivo anterior. (Figura A. 12)

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
Amatriz.m	11/2/2021 13:04	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	28/9/2021 1:02	MATLAB Code	5 KB
MLPmain.m	16/12/2021 19:54	MATLAB Code	5 KB
NNmatlab.m	20/8/2021 21:51	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB
sigmoid.m	25/10/2018 16:16	MATLAB Code	2 KB

Figura A. 12 Archivo que contiene función propia de Matlab

SVM

Para el algoritmo Support Vector Machine se abre la carpeta 03SVM y se abre el archivo mainSVM.m donde está el programa principal. (Figura A. 13)

ajustarsvm.m	18/11/2021 0:50	MATLAB Code	2 KB
alejamiento.m	18/11/2021 0:48	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	16/12/2021 23:18	MATLAB Code	4 KB
mainSVM.m	16/12/2021 23:17	MATLAB Code	3 KB
probarSVM.m	18/11/2021 0:50	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB
SVMmatlab.m	17/12/2021 0:31	MATLAB Code	2 KB

Figura A. 13 Carpeta que almacena programa MLP

Si se quiere escoger un dataset de entrenamiento diferente se cambia el nombre del dataset de acuerdo a las Tablas Tabla 1.1Tabla 1.2, en la Figura A. 14 se puede observar que en la línea 5 y si se quiere cambiar el dataset de prueba se cambia la línea 13 del código por un nombre valido de la Tabla 1.3. Se debe recalcar que los datasets de entrenemiento y de prueba deben tener el mismo número de clases.

mainSVM.m

```
1    % Principal
2 -  clc, clear, close all;
3    %% Datos de entrenamiento
4 -  load('Entrenamiento.mat');
5 -  entrenamiento = train4c200;
6 -  entrenamiento(:,1:end-1)=zscore(entrenamiento(:,1:end-1))
7 -  XY_Train = entrenamiento(:,1:end-1).';
8 -  etiqueta_Train = entrenamiento(:,end).';
9
10   %% Datos de prueba
11 -  load('Prueba.mat');
12 -  prueba = test4c;
13 -  Xp = prueba(:,1:end-1);
14 -  prueba(:,1:end-1)=zscore(prueba(:,1:end-1));
```

Figura A. 14 Líneas que se deben cambiar en programa SVM

Cuando se tienen los datasets correctos se corre el programa y se obtiene las Figura A. 15.

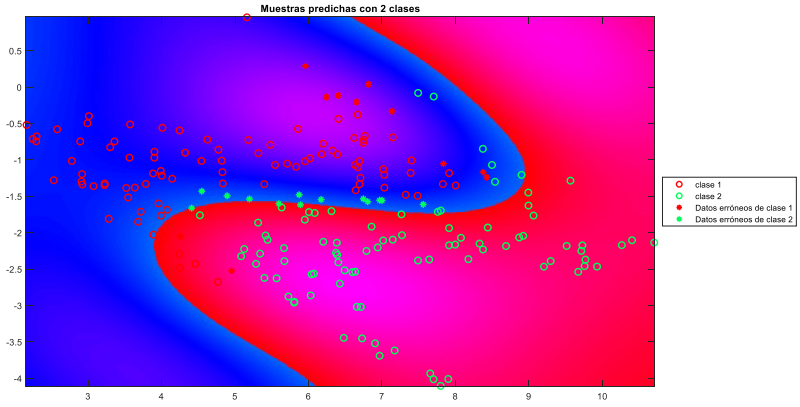


Figura A. 15 Gráfica resultante de correr programa SVM

En la Figura A. 16 se puede ver la matriz de confusión resultante.

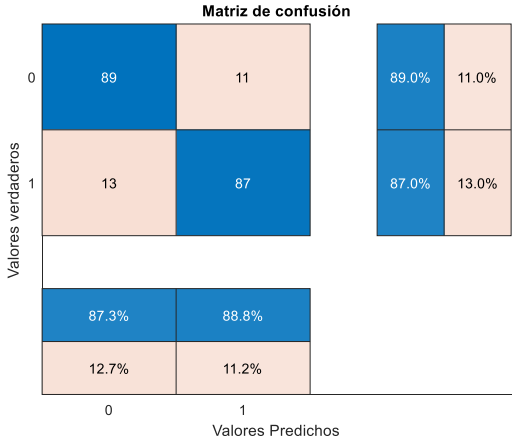


Figura A. 16 Matriz de confusión de SVM

Si se quiere comparar con un algoritmo de las herramientas de Matlab se puede elegir el archivo SVMmatlab y se puede correr de la misma manera que se hizo para el archivo anterior.

ajustarsvm.m	18/11/2021 0:50	MATLAB Code	2 KB
alejamiento.m	18/11/2021 0:48	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	16/12/2021 23:18	MATLAB Code	4 KB
mainSVM.m	16/12/2021 23:17	MATLAB Code	3 KB
probarSVM.m	18/11/2021 0:50	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB
SVMmatlab.m	17/12/2021 0:31	MATLAB Code	2 KB

Figura A. 17 Archivo que contiene función propia de Matlab

DECISION TREE

Para el algoritmo Decision Tree se abre la carpeta 04DT y se abre el archivo DTmain.m donde está el programa principal.

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
DTmain.m	28/9/2021 0:23	MATLAB Code	3 KB
DTMatlab.m	16/12/2021 22:43	MATLAB Code	2 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	13/7/2021 21:31	MATLAB Code	4 KB
hacerArbol.m	27/7/2021 20:10	MATLAB Code	3 KB
probarArbol.m	16/12/2021 0:50	MATLAB Code	1 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB
test_data.mat	12/3/2021 11:32	MATLAB Data	1 KB

Figura A. 18 Carpeta que almacena programa DT

Si se quiere escoger un dataset de entrenamiento diferente se cambia el nombre del dataset de acuerdo a las Tablas Tabla 1.1Tabla 1.2, como se observa en la Figura A. 19 en la línea 5 y si se quiere cambiar el dataset de prueba se cambia la línea 13 del código por un nombre valido de la Tabla 1.3. Se debe recalcar que los datasets de entrenemiento y de prueba deben tener el mismo número de clases.

DTmain.m

```
1   % DTmain
2   clc; clear, close all;
3   %% Datos de entrenamiento
4   load('Entrenamiento.mat');
5   datos_entrenar = train2c500;
6   XY_Train = datos_entrenar(:,1:2);
7   XY_Train = XY_Train.';
8   etiqueta_Train = datos_entrenar(:,3)+1;
9   etiqueta_Train = etiqueta_Train.';
10
11  %% Datos de prueba
12  load('Prueba.mat');
13  datos_prueba = test2c;
14  XY_Test = datos_prueba(:,1:2);
15  xT = XY_Test.';
16  etiqueta_Test = datos_prueba(:,3)+1;
17  etiqueta_Test = etiqueta_Test.';
```

Figura A. 19 Líneas que se deben cambiar en programa DT

Cuando se tienen los datasets correctos se corre el programa y se obtiene la Figura A. 20.

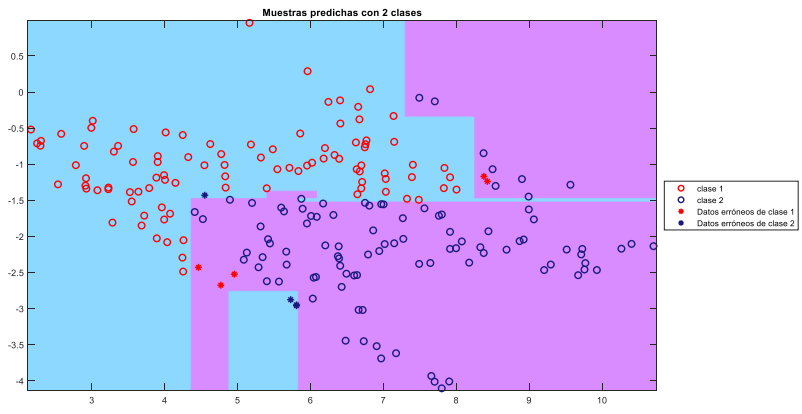


Figura A. 20 Gráfica resultante de correr programa DT

En la Figura A. 21Figura A. 10 se puede ver la matriz de confusión resultante .

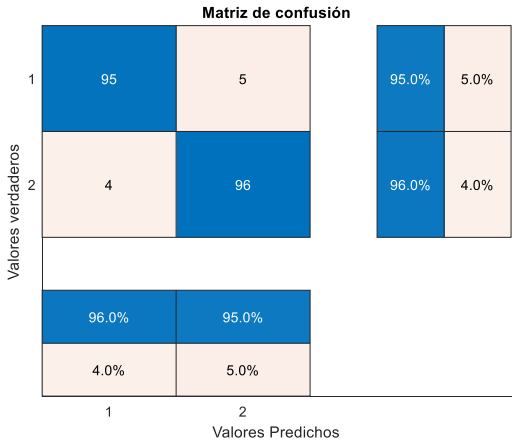


Figura A. 21 Matriz de confusión de DT

Si se quiere comparar con un algoritmo de las herramientas de Matlab se puede elegir el archivo DTmatlab y se puede correr de la misma manera que se hizo para el archivo anterior. (Figura A. 22)

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
DTmain.m	28/9/2021 0:23	MATLAB Code	3 KB
DTMatlab.m	16/12/2021 22:43	MATLAB Code	2 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	13/7/2021 21:31	MATLAB Code	4 KB
hacerArbol.m	27/7/2021 20:10	MATLAB Code	3 KB
probarArbol.m	16/12/2021 0:50	MATLAB Code	1 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB
test_data.mat	12/3/2021 11:32	MATLAB Data	1 KB

Figura A. 22 Archivo que contiene función propia de Matlab

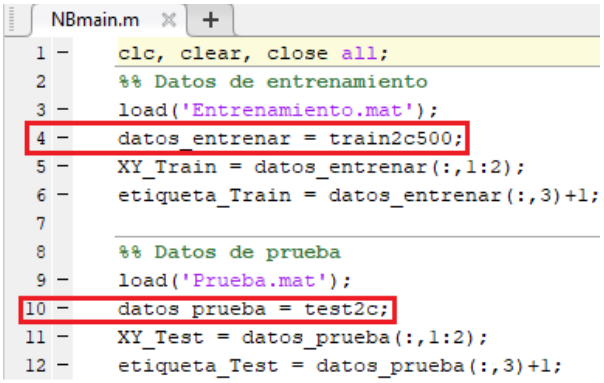
NAIVE BAYES

Para el algoritmo Naive Bayes se abre la carpeta 05 y se abre el archivo NBmain.m donde está el programa principal. (Figura A. 23)

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	14/12/2021 16:22	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	10/8/2021 0:01	MATLAB Code	4 KB
NBmain.m	14/12/2021 16:21	MATLAB Code	4 KB
NBmatlab.m	10/8/2021 0:01	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB

Figura A. 23 Carpeta que almacena programa NB

Si se quiere escoger un dataset de entrenamiento diferente se cambia el nombre del dataset de acuerdo a las Tablas Tabla 1.1Tabla 1.2, en Figura A. 24 se observa en la línea 4 y si se quiere cambiar el dataset de prueba se cambia la línea 10 del código por un nombre valido de la Tabla 1.3. Se debe recalcar que los datasets de entrenemiento y de prueba deben tener el mismo número de clases.



NBmain.m

```
1  clc, clear, close all;
2  %% Datos de entrenamiento
3  load('Entrenamiento.mat');
4  datos_entrenar = train2c500;
5  XY_Train = datos_entrenar(:,1:2);
6  etiqueta_Train = datos_entrenar(:,3)+1;
7
8  %% Datos de prueba
9  load('Prueba.mat');
10 datos_prueba = test2c;
11 XY_Test = datos_prueba(:,1:2);
12 etiqueta_Test = datos_prueba(:,3)+1;
```

Figura A. 24 Líneas que se deben cambiar en programa NB

Cuando se tienen los datasets correctos se corre el programa y se obtiene la Figura A. 25.

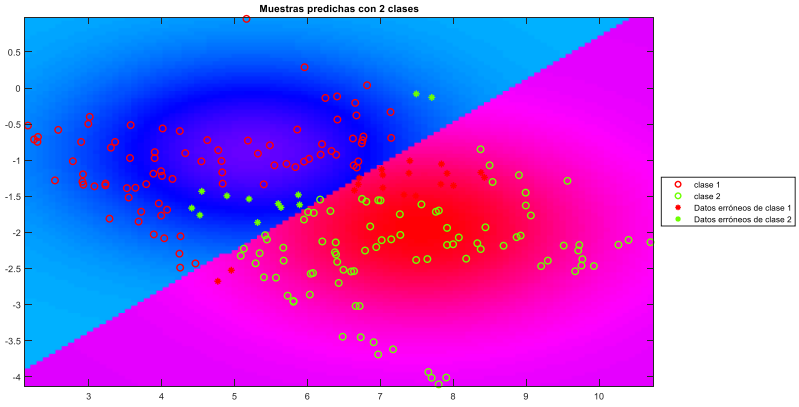


Figura A. 25 Gráfica resultante de correr programa NB

En la Figura A. 26 se puede ver la matriz de confusión resultante y en Figura A. 27 se observa resultante de salidas de algoritmo.

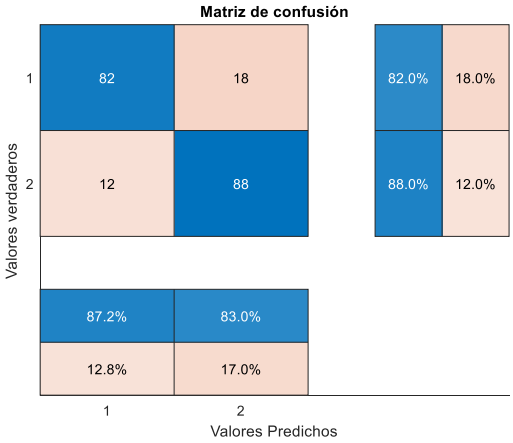


Figura A. 26 Matriz de confusión de NB

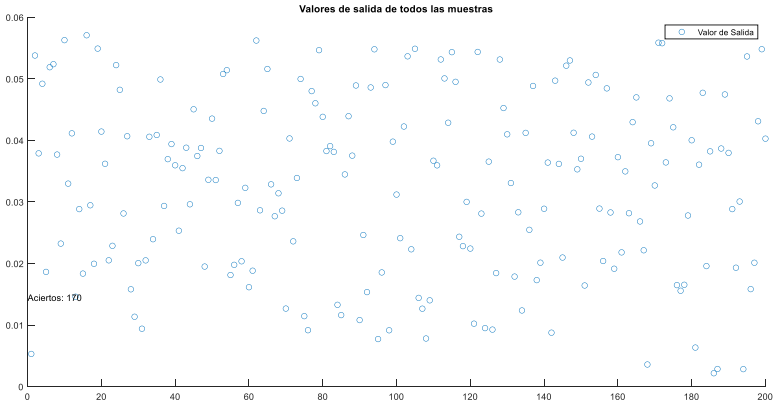


Figura A. 27 Valores de salidas de NB

Si se quiere comparar con un algoritmo de las herramientas de Matlab se puede elegir el archivo NBmatlab y se puede correr de la misma manera que se hizo para el archivo anterior. (Figura A. 28)

Figura A. 28 Archivo que contiene función propia de Matlab

KNN

Para el algoritmo K-Nearest Neighbor se abre la carpeta 06KNN y se abre el archivo mainKNN.m donde está el programa principal. (Figura A. 29)

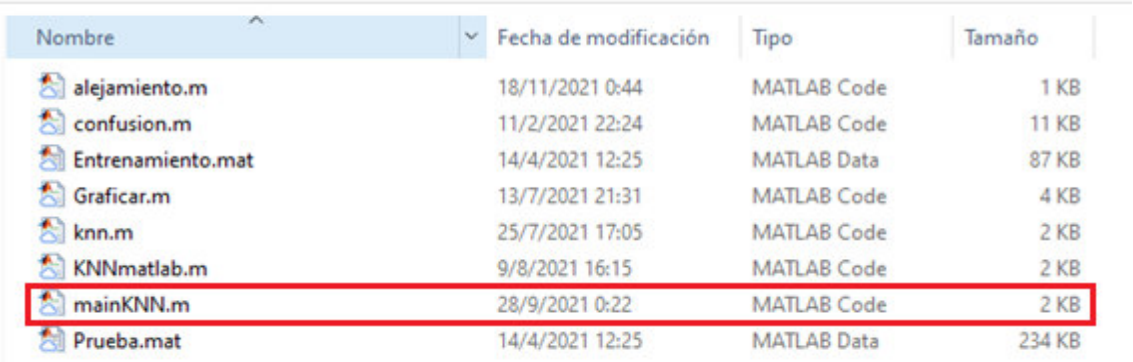


Figura A. 29 Carpeta que almacena programa KNN

Si se quiere escoger un dataset de entrenamiento diferente se cambia el nombre del dataset de acuerdo a las Tablas Tabla 1.1Tabla 1.2, en la Figura A. 30 se puede ver que en la línea 5 y si se quiere cambiar el dataset de prueba se cambia la línea 12 del código por un nombre valido de la Tabla 1.3. Se debe recalcar que los datasets de entrenemiento y de prueba deben tener el mismo número de clases.

```
mainKNN.m
1   %% main K-NN
2   %% Datos de entrenamiento
3 - clc, clear; close all;
4 - load('Entrenamiento.mat');
5 - datos_entrenar = train2c100;
6 - XY_Train = datos_entrenar(:,1:end-1);
7 - etiqueta_Train = datos_entrenar(:,end);
8 - k = 3;
9
10  %% Datos de prueba
11 - load('Prueba.mat');
12 - datos_prueba = test2c;
13 - XY_Test = datos_prueba(:,1:end-1);
14 - etiqueta_Test = datos_prueba(:,end);
```

Figura A. 30 Líneas que se deben cambiar en programa KNN

Cuando se tienen los datasets correctos se corre el programa y se obtiene la Figura A. 31.

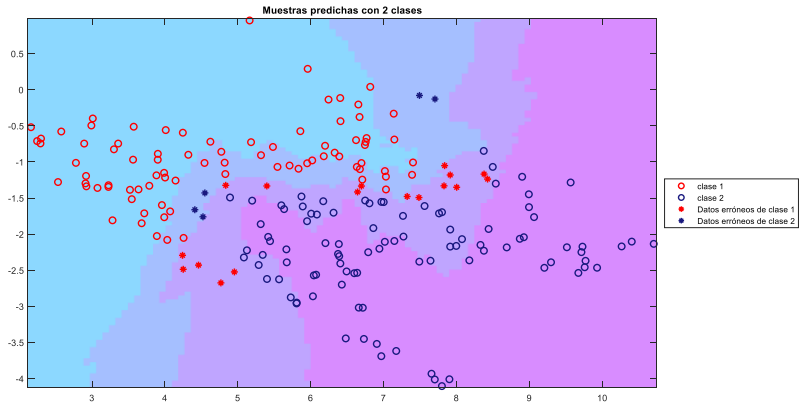


Figura A. 31 Gráfica resultante de correr programa KNN

En la Figura A. 32 se puede ver la matriz de confusión resultante y en Figura A. 33 se observa resultante de salidas de algoritmo.

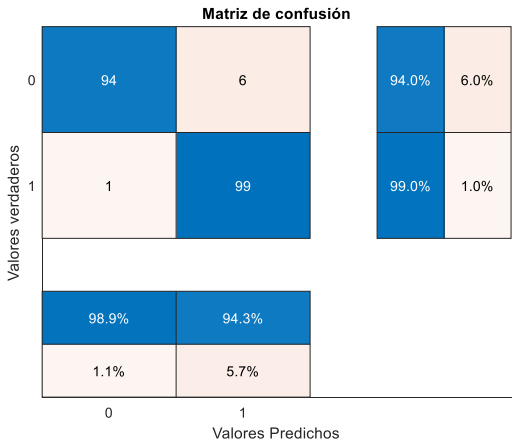


Figura A. 32 Matriz de confusión de KNN

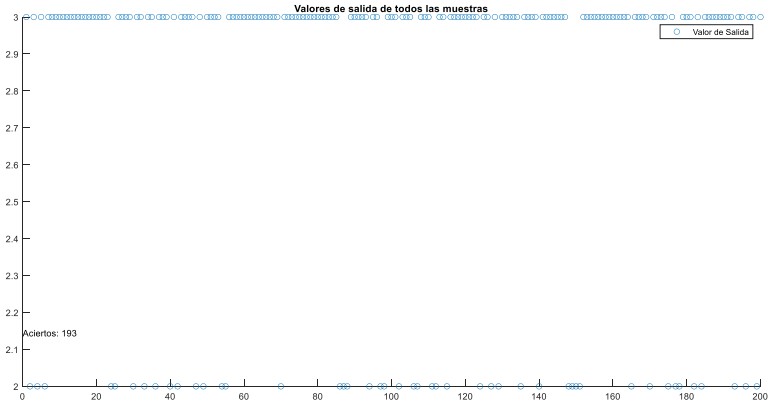


Figura A. 33 Valores de salidas de KNN

Si se quiere comparar con un algoritmo de las herramientas de Matlab se puede elegir el archivo KNNmatlab.m y se puede correr de la misma manera que se hizo para el archivo anterior. (Figura A. 34)

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	13/7/2021 21:31	MATLAB Code	4 KB
knn.m	25/7/2021 17:05	MATLAB Code	2 KB
KNNmatlab.m	9/8/2021 16:15	MATLAB Code	2 KB
mainKNN.m	28/9/2021 0:22	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB

Figura A. 34 Archivo que contiene función propia de Matlab

PNN

Para el algoritmo PNN se abre la carpeta 07PNN y se abre el archivo PNN.m donde está el programa principal. (Figura A. 35)

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
Amatriz.m	11/2/2021 13:04	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	18/8/2021 22:53	MATLAB Code	4 KB
PNN.m	14/12/2021 16:33	MATLAB Code	4 KB
PNNmatlab.m	6/9/2021 19:16	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB

Figura A. 35 Carpeta que almacena programa PNN

Si se quiere escoger un dataset de entrenamiento diferente se cambia el nombre del dataset de acuerdo a las Tablas Tabla 1.1Tabla 1.2, en la Figura A. 36 se puede observar la línea 5 y si se quiere cambiar el dataset de prueba se cambia la línea 11 del código por un nombre valido de la Tabla 1.3. Se debe recalcar que los datasets de entrenemiento y de prueba deben tener el mismo número de clases.

```
PNN.m
1   %% Red probabilistica
2   %% Datos de entrenamiento
3 - clc, clear, close all;
4 - load('Entrenamiento.mat');
5 - datos_entrenar = train4c3m;
6 - XY_Train = datos_entrenar(:,1:2);
7 - XY_Train = zscore(XY_Train);
8 - etiqueta_Train = datos_entrenar(:,3)+1;
9   %% Datos Prueba
10 - load('Prueba.mat');
11 - datos_prueba = test4c;
12 - XY_Test = datos_prueba(:,1:2);
13 - xy_T = zscore(XY_Test);
```

Figura A. 36 Líneas que se deben cambiar en programa PNN

Cuando se tienen los datasets correctos se corre el programa y se obtiene la Figura A. 37.

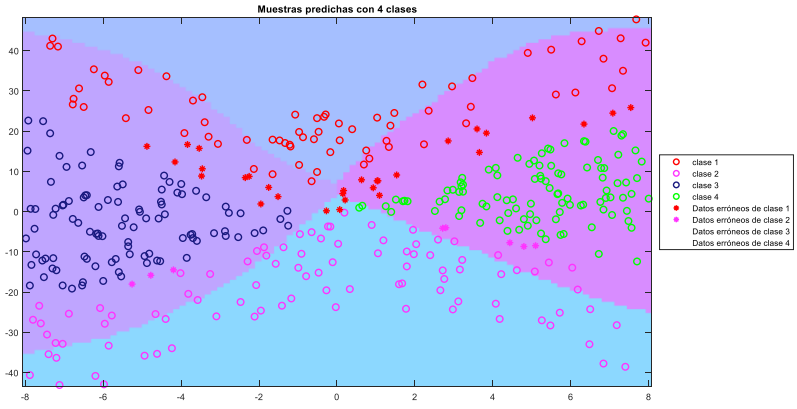


Figura A. 37 Gráfica resultante de correr programa PNN

En la Figura A. 38 se puede ver la matriz de confusión.

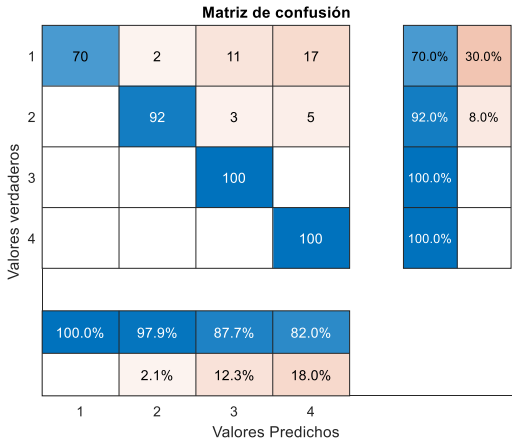


Figura A. 38 Matriz de confusión de PNN

Si se quiere comparar con un algoritmo de las herramientas de Matlab se puede elegir el archivo PNNmatlab y se puede correr de la misma manera que se hizo para el archivo anterior. (Figura A. 39)

Nombre	Fecha de modificación	Tipo	Tamaño
alejamiento.m	18/11/2021 0:44	MATLAB Code	1 KB
Amatriz.m	11/2/2021 13:04	MATLAB Code	1 KB
confusion.m	11/2/2021 22:24	MATLAB Code	11 KB
Entrenamiento.mat	14/4/2021 12:25	MATLAB Data	87 KB
Graficar.m	18/8/2021 22:53	MATLAB Code	4 KB
PNN.m	14/12/2021 16:33	MATLAB Code	4 KB
PNNmatlab.m	6/9/2021 19:16	MATLAB Code	2 KB
Prueba.mat	14/4/2021 12:25	MATLAB Data	234 KB

Figura A. 39 Archivo que contiene función propia de Matlab

ORDEN DE EMPASTADO