Desarrollo de un framework de clasificación de patrones

Ignacio Ramírez, Andrés Alcarraz, Alicia Fernández y André Fonseca de Oliveira
Instituto de Ingeniería Eléctrica - Facultad de Ingeniería
Universidad de la República Oriental del Uruguay
{ nacho, alcarraz, alicia, andre } @fing.edu.uy

24 de octubre de 2002

Resumen

El presente trabajo describe la arquitectura de un paquete desarrollado para sistemas de reconocimiento de patrones.

Se detalla el estudio de los elementos comunes y las variabilidades existenten en diversos sistemas de reconocimiento de patrones propuestos en la literatura, utilizados en la determinación de la arquitectura del paquete.

De este análisis se ha diseñado un paquete que define en forma fija la lógica, estructura e implementación de las partes identificadas como comunes a los sistemas de reconocimiento de patrones analizados, dejando incompletas las particularidades de estos.

De este modo el producto permite desarrollar aplicaciones que se adapten a cada caso, posibilitando al desarrollador concentrarse solamente en las variaciones (modelos, algoritmos, parámetros).

Al final del trabajo se describe la prueba del paquete desarrollado, utilizándolo como base para el desarrollo de un sistema de clasificación de patrones sencillo cuyo fin es el agrupamiento de maderas por colores para su uso en mueblería.

1. Introducción

Este trabajo surge como respuesta a la necesidad de centralizar los conceptos y conocimientos generales a toda aplicación de reconocimiento de patrones (RP) en un paquete de software reutilizable, surgida en el contexto académico del IIE[1] y en particular dentro del GTI[2], para complementar al resto de los paquetes que hasta el momento se han desarrollado en dicha institución.

1.1. Herramientas existentes

En el contexto actual no se tiene conocimiento de que existan herramientas con características similares a las deseadas para este producto, y que se adapten a las necesidades del IIE.

1.2. Objetivos

En resumen, las metas que este paquete pretende alcanzar son:

- Facilitar el desarrollo de aplicaciones de clasificación de patrones, suministrando una arquitectura prefabricada con sus conceptos y algoritmos más comunes.
- Definir un mecanismo único, pronto para utilizar, mediante el cual configurar y almacenar sistemas de clasificación de patrones.
- Proveer una capa de abstracción superior sobre los paquetes y conjuntos de algoritmos ya implementados en el IIE para el tratamiento de señales, en particular imágenes.

[1] Instituto de Ingeniería Eléctrica
[2] Grupo de Tratamiento de Imágenes

1.3. Vista general

En terminología de software, el paquete define un *framework* de clasificación de patrones, es decir, un marco de trabajo genérico cuyo fin es facilitar el desarrollo de aplicaciones concretas de clasificación de patrones, centralizando los conceptos y algoritmos comunes a la familia definida por este tipo de aplicaciones.

Además de la implementación de los algoritmos y conceptos generales inherentes a la Teoría de Reconocimiento de Patrones (en particular la Clasificación de Patrones), el paquete define un conjunto de facilidades para uniformizar el mecanismo de *persistencia*, codificación y transmisión de los datos dentro de un sistema de clasificación; y para consultar, definir y modificar los parámetros de los algoritmos involucrados.

Junto con las clases básicas del paquete se incluyen implementaciones de referencia para algunos algoritmos de clasificación populares como ser el Algoritmo de Bayes o el K-Means.

Finalmente, como resultado de las especificaciones del proyecto, se incluye especialmente un subframework para el desarrollo de sistemas basados en *redes neuronales artificiales*. Este ultimo es mucho mas que un agregado al paquete, constituyendo parte importante de el y pudiendo ser utilizado en otros contextos no relacionados con el reconocimiento de patrones, como por ejemplo compresión, control automático, etc.

1.4. Modo de uso

A diferencia de otro tipo de paquetes de software, el trabajo aquí presentado no provee una *aplicación funcional* lista para utilizar. Los usuarios de un *framework* son aquellos desarrolladores que deseen aprovechar los algoritmos, estructuras y conceptos de este paquete para crear aplicaciones concretas y funcionales diseñadas a medida para un problema particular.

El *framework* se compone de un conjunto de clases comunes a todo sistema, funcionales y que usualmente no deben ser especializadas, denominadas *frozen spots*, junto con una o mas clases abstractas que *deben* ser especializadas para cada caso particular, denominadas *hotspots*.

Quien desarrolle una aplicación basada en este paquete deberá entonces definir una o mas implementaciones concretas de los *hotspots* que ajusten a sus necesidades.

1.5. Ventajas adicionales

Además de facilitar y catalizar el desarrollo de nuevas aplicaciones, mediante su uso y propia evolución el framework sirve como contenedor de la base de conocimientos sobre la aplicación de la teoría del reconocimiento de patrones.

1.6. Estructura del documento

En la sección 2 se describe el proceso de análisis previo al diseño del paquete, que comprende al estudio de las principales características de los sistemas de reconocimiento estudiados. La sección 3 detalla el diseño del sistema. Luego en la sección 4 se describen algunos detalles de la implementación (lenguaje de programación, etc.).

En la segunda parte del documento, comenzando en la sección 5 , se habla específicamente del subframework de redes neuronales artificiales que forma parte del framework general.

En la sección 6 se describe una aplicación sencilla de clasificación, creada para probar los distintos aspectos del paquete desarrollado. Finalmente en la sección 7 se comentan los resultados obtenidos, las conclusiones en la 8 y finalmente algunos lineamientos sobre trabajos futuros en la sección 10.

2. Análisis

El principal objetivo de este trabajo fue lograr un producto flexible y reutilizable, que sirviera para desarrollar diversos tipos de sistemas.

Para esto, se pasó por una etapa de análisis en la cual se estudiaron distintos tipos de sistemas de reconocimiento de patrones. El foco del análisis fue identificar los aspectos comunes a todos ellos, y las particularidades que los distinguen entre sí.

Los tipos de sistemas estudiados se clasificaron según los paradigma (o modelos) teóricos sobre los cuales se basan, algoritmos que utilizan, modo de aprendizaje y operación, etc.

Dentro de los paradigmas vistos se incluyen los estadísticos, los estructurales, los KBS y los lógico-combinatorios, prestando especial atención a los primeros ya que constituyen la mayoría de los sistemas referidos en la literatura.

En cuanto a algoritmos se estudiaron sistemas paramétricos como el clasificador de Bayes o el K-Means, y no paramétricos, en especial la familia de las redes neuronales artificiales.

Por último se buscó, en cada familia, ejemplos de sistemas no entrenables, entrenables y adaptivos.

Referencias

Durante la etapa de análisis se estudiaron algoritmos como el de *Bayes*, tratado en libros como el de Duda y Hart [1] y el de Tou y Gonzales [2]. Algoritmos como el *K-Means*, *K-NEarest Neighbour*, así como los lógico combinatorios como el de *Testores Típicos* o el *Class* fueron tomados del libro de Shulcloper, Arenas y Trinidad [3] durante el curso dictado por el profesor Manuel Lazo Cortés del ICMF (Cuba). Las referencias utilizadas para los algoritmos basados en redes neuronales artificiales son descritas más adelante en la sección 5.

Además de los algoritmos específicos, el estudio general sobre sistemas de reconocimiento fue completado con trabajos como el de Cortijo [4], Crida [5], Pandya [6]. De este último fueron tomadas también las nociones básicas sobre sistemas de reconocimento estructurales. La información sobre los KBS (Knowledge Based Systems), no tenidos en cuenta originalmente, fue tomada a partir del sitio de Richard O. Duda [7] y recolectada de diversos lugares de Internet, recomendando al sitio anteriormente mencionado como punto de partida. Este sitio posee además información teórica y referencias a otros lugares relacionados con el RP.

2.1. Conceptos comunes (invariantes)

El primer aspecto común a todos los sistemas vistos es que admiten una representación de su funcionamiento en etapas bien distinguidas: adquisición (o sensado) de datos, extracción de patrones (o caracterización) y actuación (en particular, clasificación).

También surjen como invariantes los tipos de datos que fluyen de una etapa hacia la siguiente.

En este trabajo se focaliza en la etapa de clasificación, los datos de entrada a dicha etapa y los resultados obtenidos a la salida.

En este contexto se vió que, independientemente del significado que cada paradigma da a los patrones, su función y contenido pueden ser codificados en una misma estructura. Es decir, un patrón siempre podrá ser visto como un conjunto de características, sean símbolos o magnitudes. También, dependiendo del paradigma, los patrones obtenidos en un mismo proceso pueden ser de tamaño variable o fijo. Esto también puede unificarse, asumiendo el primer caso por defecto.

Durante el entrenamiento de un sistema, mas allá del paradigma o algoritmo utilizado, la información es suministrada en forma de ejemplos que representan prototipos de los tipos de objetos o entidades a reconocer. La función de un ejemplo como unidad de información de entrenamiento es algo que también se identificó como independiente del sistema en cuestión. Un modelo suficientemente útil al que se llegó es el de un ejemplo conformado por un patrón representando al prototipo mas un conjunto de información adicional.

Si el sistema es de clasificación, y el algoritmo de clasificación utilizado es supervisado, la información adicional toma la forma de la clase correcta a la que el prototipo pertenece según un experto (artificial o humano).

Asociado a un ejemplo también aparece en algunos sistemas el concepto de peso informacional del ejemplo, valor que pondera qué tanta información aporta cada ejemplo al entrenamiento, o bien con qué certeza el resultado de referencia es obtenido, para sistemas supervisados.

Tanto la información adicional como el peso informacional no aparecen en todos los sistemas vistos, pero con su inclusión no se pierde generalidad, siendo ignorada esta información por aquellos sistemas que no la utilicen.

La salida de un sistema de clasificación puede ser expresada en una forma lo suficientemente general como para abarcar los sistemas vistos. Concretamente, si un sistema de clasificación conoce N clases, a lo sumo podrá inferir el grado de pertenencia del patrón clasificado a cada una de ellas. Por ejemplo, si el sistema utiliza lógica borrosa, los grados de pertenencia pueden ser continuos en el rango [0,1], mientras que en sistemas basados en teoría de conjuntos clásica, los valores admisibles son 0 y 1.

Las clases también son modeladas en forma única, asignando "etiquetas" numéricas a cada una de ellas. La posible descripción de cada clase se deja en manos de la aplicación concreta a implementar.

Por último, la principal invariante es la que refiere a la *interfaz* de la etapa de clasificación, es decir, su modelo de "caja negra". En este aspecto, todos los clasificadores vistos cumplen con una operación de *clasificación* que recibe un patrón y produce un resultado. También se puede definir una operación de *entrenamiento*

La operación de entrenamiento pudo ser unificada para todos los sistemas vistos al dividirse en dos suboperaciones consecutivas: *recolección de la información* suministrada por los ejemplos, y *adaptación del sistema* en base a la información almacenada. A estas operaciones se las identificó como *recolección* y *adaptación*.

Estas operaciones pueden ser definidas sea el sistema entrenable, adaptivo o incluso no entrenable. Por ejemplo, para este último caso, ambas operaciones simplemente se ignoran.

2.2. Variabilidades

La principal variabilidad del sistema reside en el *algoritmo utilizado para clasificar*, la información necesaria para ello y el algoritmo de entrenamiento para sintetizar esa información.

3. Diseño

El diseño parte de los conceptos centrales identificados en la mayoría de los sistemas de clasificación: *clasificador, patrón, cluster, resultado, clase, ejemplo.* El primer criterio adoptado fue la representación de cada uno de estos conceptos centrales por clases.

3.1. Clases principales

3.1.1. Patrones

La clase `Pattern` define el concepto general de patrón como un vector real de características de tamaño variable. Esta representación se acomoda especialmente a los sistemas de clasificación estadísticos, que admiten una representación *vectorial* en un *espacio de características*. En contraparte no es del todo adecuada para sistemas cuyas características son *etiquetas* o *símbolos* como podría ser el caso de clasificadores *sintácticos*[3], *lógico combinatorios* [3] o *KBS* [9], cita tomada del sitio web de Richard O. Duda [7], mas que nada en lo que respecta a eficiencia en el almacenamiento. Esta decisión se fundamenta en el hecho de que la mayoría de los sistemas de clasificación se enmarcan en el paradigma estadístico y en la necesidad de evitar sobrecargas excesivas en tiempos de ejecución para clases de uso intensivo como *Pattern*. El resultado es una clase *Pattern* de poco peso y sobre todo concreta, evitando así la sobrecarga debida al mecanismo de *polimorfismo dinámico* de lenguajes como C++.

3.1.2. Ejemplos de entrenamiento

`Example` encapsula un ejemplo de entrenamiento formado por un *Pattern* proveniente de un prototipo de entrenamiento y eventualmente un *resultado de clasificación de referencia* provisto por un *experto* para el uso en algoritmos de clasificación supervisados.

[3] Ver Chomsky [8], cita tomada del libro de Pandya [6]

Esta implementación define además un *peso informacional* para cada ejemplo, que puede tomar valores entre 0 (no aporta información) y 1 (máxima información).

3.1.3. Resultados

`Result` representa un resultado genérico de clasificación. En el diseño realizado, éste se expresa como una lista de pares *clase-grado de pertenencia*, con un par para cada clase definida.

El clasificador que produce un resultado, lo hace creando una instancia de esta clase y asignando al patrón clasificado un grado de pertenencia a cada clase conocida.

La forma en que esta clase fue definida permite abarcar resultados de clasificación de diversos tipos de sistemas desde Bayes hasta sistemas de *lógica borrosa* como el [10].

Las clases son representadas por *identificadores numéricos*, reservando el "ID" de valor 0 para la *clase de rechazo*, utilizada por muchos algoritmos como aquella a la que se asignan los patrones de no haber clasificación satisfactoria dentro de las otras clases existentes.

3.2. Classifier

El principal *hotspot* es la clase abstracta `Classifier` y la interfaz que define sus operaciones.

La principal operación a definir es el propio algoritmo de clasificación, dada por método `classify()`.

Además, si el clasificador a diseñar requiere *entrenamiento* deben implementarse o bien los métodos `train()` y `batchTrain()` o bien el par de métodos `adapt()` y `learn()`.

`train()` y `batchTrain()` se utilizan para *entrenar* o *reentrenar* a un sistema de clasificación a partir de uno (`train()`) o varios (`batchTrain()`) ejemplos de entrenamiento. Este es el enfoque mas clásico de entrenamiento de sistemas de clasificación.

Por otro lado se define también el par `learn()`-`adapt()`, previsto para algoritmos no sólo *entrenables* sino *adaptivos*, pudiéndose utilizar en cualquiera de las dos situaciones. La idea es que `learn()` colecta información de aprendizaje, de a un ejemplo por vez. `adapt()` sintetiza toda la información recolectada por `learn()` y efectúa la adaptación del sistema según ella.

3.2.1. train() y batchTrain() vs. learn() y adapt()

En este contexto se considera como *entrenables* a aquellos sistemas que requieran un conjunto de ejemplos (o prototipos) de entrenamiento para definir sus parámetros, sin implicar esto que dichos sistemas sean capaces de adaptarse a cambios en la naturaleza del problema durante su utilización. Llegado este último caso, estos sistemas pueden ser a lo sumo *reentrenados* con un nuevo conjunto de datos actualizado.

Por su parte, los sistemas *adaptivos* son aquellos cuya estructura admite una modificación *incremental* de su estado interno para adaptarse al problema, incluso en tiempo real como podría ser en una línea de producción.

`learn()` y `adapt()` permiten este comportamiento y dan lugar a optimizaciones del tipo que puede verse en algoritmos tipo *batch*, permitiendo a la vez su uso en tiempo real. Es posible también definir el proceso de entrenamiento en función de estos métodos, aunque en algunos casos sea poco práctico o conceptualmente forzado. De hecho, `train()` y `batchTrain()` poseen implementaciones por defecto basadas en `learn()` y `adapt()` de forma de reutilizar código al máximo.

3.2.2. Interfaz de persistencia y configuración

`Classifier` implementa por defecto las interfaces `Encodable` y `Configurable` para poder ser propiamente almacenado y configurado. Ambas interfaces serán descritas más adelante en este documento.

3.2.3. Matriz de confusión

Como mecanismo estándar de evaluación de desempeño para algoritmos supervisados se provee el método `getConfusionMatrix()` que permite, en base a un conjunto de ejemplos de prueba, analizar el

desempeño de un clasificador dado en base a la comparación de los resultados producidos por él y los resultados esperados, almacenados en los ejemplos de prueba.

La matriz de confusión se confecciona de la siguiente manera:

1. Se define el tamaño de la matriz como $N \times N$, siendo N la cantidad de clases definidas, y se asocia un índice a cada clase existente, secuencialmente de 1 a N.

2. Para cada ejemplo de prueba:

   a) se determina i como el índice de la clase *correcta* contenida en el ejemplo.

   b) se determina j como el índice de la clase obtenida como resultado de la clasificación del ejemplo.

   c) se incrementa en 1 la casilla i, j.

De esta forma, todos los elementos de una fila corresponden a una misma clase *correcta*, mientras que los de una columna son los que el clasificador ubicó en la misma clase.

En la representación utilizada en las tablas, la matriz es de $N + 1 \times N + 1$. La última fila de cada tabla contiene el *índice de confiabilidad* del clasificador, que indica que tan probable es que "cuando el clasificador decida la clase dada, el resultado verdadero sea efectivamente ese". El cálculo para este índice es:

$$confiabilidad_j = \frac{c_{jj}}{\sum_{i=1}^{N} c_{ij}}$$

La última columna, por su parte, indica el *índice de confusión* e indica "que tan bien clasifica el sistema a los elementos de la clase". Su cálculo viene dado por la expresión

$$confusion_j = \frac{c_{ii}}{\sum_{j=1}^{N} c_{ij}}$$

Por último, el elemento $[N + 1, N + 1]$ contiene un índice de desempeño global, que se calcula como

$$perf = \frac{\sum_{i=1}^{N} c_{ii}}{\sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij}}$$

Naturalmente, otros mecanismos de evaluación del desempeño pueden ser definidos por los algoritmos particulares que implementen la interfaz de `Classifier`.

3.3. Configuración

Hay otras funciones que un sistema de clasificación real debe poseer además de clasificar y ser entrenado. En particular los algoritmos de clasificación, sobre todo para la etapa de entrenamiento, suelen definir una serie de parámetros que los gobiernan (por ejemplo el *factor de aprendizaje* presente en muchas redes neuronales adaptivas).

Para estas dos operaciones esenciales se definen las clases `Configurable` y `Configuration` y las interfaces `Encodable` e `IOManager`. Las primeras dos definen un mecanismo común y una estructura para el almacenamiento, consulta y modificación de parámetros definidos por un clasificador, sus algoritmos y subalgoritmos. `Encodable` e `IOManager` por su parte definen las interfaces que se deben implementar para que un sistema de clasificación y todas las clases que definen su estado y comportamiento puedan exportar su información interna de modo que pueda ser almacenada. La interfaz `Encodable` define cómo los distintos objetos deben exportar su información, en cambio `IOManager` define la interfaz que se debe implementar para definir un formato de exportación de la información, por ejemplo a un archivo XML.

El punto saliente del mecanismo de configuración es que permite consultar en tiempo de ejecución todos los parámetros existentes para un sistema de clasificación sin conocerse las clases, algoritmos y/o *estrategias* involucrados. Para ello se definió un esquema similar a *DNS* en el cual cada algoritmo tiene un *dominio* asociado y sus nodos son parámetros o nombres de *subdominios* definidos por *subalgoritmos*.

Para clarificar este concepto, supóngase un sistema de clasificación que utiliza el algoritmo *LMS* (Least Medium Square) para su adaptación. El *LMS* a su vez define una serie de parámetros propios como el *factor de aprendizaje* o *factor de momentum*. Por último el sistema define sus propios parámetros, por ejemplo, *error mínimo* o una opción para *mezclar los patrones de entrenamiento*. Al consultarse la lista de parámetros configurables para el sistema se obtendría la siguiente lista:

- classifier.minerr
- classifier.mezclar
- classifier.lms.lr
- classifier.lms.momentum

Los "dominios" definidos son en este caso "classifier" para el sistema de clasificación general y "lms" para el algoritmo lms. Notar que si el algoritmo de adaptación fuera otro (por ejemplo Kalman), la consulta devolvería la lista de parámetros correspondiente a ese caso.

3.4. Persistencia

Además, es esencial poder almacenar el estado de un sistema de clasificación entrenado y listo para ser utilizado.

Las interfaces `Encodable` e `IOManager` definen las operaciones que se deben implementar para que un sistema de clasificación y todas las clases que definen su estado y comportamiento puedan exportar su información interna de modo que pueda ser almacenada.

La interfaz `Encodable` define cómo los distintos objetos deben exportar su información.

`IOManager` define la interfaz que se debe implementar para definir un formato de exportación de la información, por ejemplo a un archivo XML.

`Encodable` utiliza el estándar *DOM* [11] como base para la codificación del estado y comportamiento de un sistema dado. Implementando esta interfaz es posible traducir esta información a diversos formatos, por defecto *XML* [12]. Tanto la codificación a *DOM* como el almacenamiento en formato *XML* dependen de la utilización de la biblioteca *Xerces* de uso público desarrollada y mantenida por el proyecto *Apache* [13].

Como nota final cabe notar que la complejidad de este diseño se debe a la necesidad, considerada importante, de independizar la forma en que se codifican internamente los datos y la forma en que se guardan y/o transmiten. Como aplicación directa de esto puede decirse que es posible, sin tocar ninguna línea del código existente, almacenar todo un sistema de clasificación en un formato totalmente nuevo (binario, ASCII, base64, etc).

3.5. Mensajes del sistema

La última de las utilidades disponibles en el framework es la clase *Logger* que permite controlar la emisión de mensajes de error, advertencias, trazado o depuración durante la ejecución según un nivel configurable en tiempo de ejecución.

3.6. Implementaciones listas para usar

Para cada *hotspot* se definieron algunas implementaciones de referencia para probar el sistema y darle cierta funcionalidad por defecto. Tal es el caso de `Classifier`, que actualmente dispone de tres implementaciones: Bayes, K-Means y NeuralNetClassifier, este último haciendo de *adaptador* para definir clasificadores basados en las redes neuronales artificiales tal como son provistas por el *subframework de redes neuronales* descrito más adelante.

4. Implementación

La implementación fue realizada en C++ que es un lenguaje orientado a objetos y a la vez permite lograr un buen desempeño en tiempo de ejecución, a costo de ser un lenguaje complejo y difícil de dominar y utilizar eficientemente.

ClasiLiptus no es un paquete autocontenido ya que depende fuertemente de otros dos paquetes de libre distribución: el *Xerces* [12] para la codificación y almancenamiento de la información inherente a los sistemas de clasificación, y la *Matrix Template Library* [14] para la representación de vectores y matrices así como para las operaciones algebraicas mas comunes (producto, producto interno, norma, matriz inversa, determinante, etc.).

5. Subframework de redes neuronales

Los algoritmos basados en *redes neuronales artificiales* son una importante rama de la familia de algoritmos de clasificación y reconocimiento de patrones y abarcan una amplia gama de variedades pensadas para resolver ciertos problemas complejos o difíciles de resolver mediante otros métodos.

Debido a esto es que el framework de clasificación dispone de un *subframework* para la implementación de redes neuronales artificiales. Este *subframework* posee además la virtud de poder ser utilizado en forma independiente de la parte de clasificación de patrones, por ejemplo en aplicaciones que no tienen que ver con el reconocimiento de patrones como la codificación de datos[4]

5.1. Análisis

Debido a la importancia atribuída a esta rama de algoritmos de reconocimiento de patrones es que se detalla aquí el análisis realizado previo al diseño de este módulo.

Al igual que para los clasificadores en general, para identificar las partes comunes y las variabilidades que caracterizan a la familia de redes neuronales artificiales (*RNA* de aquí en más), se analizó la estructura y funcionamiento de una variedad de tipos de ellas.

Referencias

Algoritmos como *Perceptron*, *Adaline*, *Backpropagation*, *Kohonen SOM*, *Hopfield*, *BAM*, *LAM*, *Counterpropagation* fueron tomados principalmente del libro de Freeman [15]. En particular, una gran cantidad de variedades de *Backpropagation* son descritas en "The Backpropagation Family Album" [16]. También se consultó el libro de Pandya y Macy [6], que describe entre otros al *LVQ* (Learning Vector Quantization). Para el estudio del *ANFIS* se consultó el trabajo de Jang [10]. Por último, fue tenida en cuenta la implementación del SNNS (Stuttgart Neural Network Simulator) [17] como referencia tanto a nivel conceptual como técnico durante el diseño y desarrollo de este módulo.

5.2. Invariantes

Las invariantes fueron tomadas en cuenta a nivel de estructura, para modelar los datos, y a nivel de interfaz, para modelar la funcionalidad.

En la primera categoría se encuentra la estructura general de una red neuronal artificial. En todos los modelos vistos, incluso en algunas definiciones "genéricas" dadas por autores como Hetch [18] (cita tomada de [6]), las redes neuronales artificiales son modeladas a nivel estructural como un *grafo dirigido* cuyos *nodos* son las unidades de procesamiento (PE) o *neuronas* y cuyos enlaces son *unidireccionales*. Cada enlace recibe un *peso* que pondera el grado de influencia de la salida unidad de origen sobre la entrada la unidad destino.

De este modo se definieron como invariantes a los elementos básicos de esta estructura, es decir, *enlace* (o conexión), y *unidad de procesamiento* (o neurona artificial), desde el punto de vista de la *conectividad*,

[4] Un ejemplo son las *LVQ - Linear Vector Quantification*, brevemente descritas en el libro de Pandya [6]

siendo el modelo computacional y operación de cada unidad una variabilidad muy importante a tener en cuenta. De este mismo análisis se definió como invariante la *estructura* de una red neuronal para cualquier tipo, y la interfaz utilizada para construir y modificar dicha estructura desde el exterior.

La interfaz principal, es decir, entrenamiento, adaptación y aplicación de la RNA, también fue fijada como invariante. También fueron fijadas las interfaces de entrenamiento, adaptación y aplicación a nivel de unidad de procesamiento. Cabe remarcar que el *fijar* una interfaz no implica *fijar* una implementación, sino restringir. En especial, las RNA como fueron modeladas reciben entradas y producen salidas de tipo *vectorial*, asumiendo en todos los casos que estas son *señales*, no *símbolos* (etiquetas, etc.) de acuerdo con la definición general vista en los textos consultados.

5.3. Variabilidades

Se distinguieron como variabilidades principales a los mecanismos de *inicialización* (o *calibración inicial* en este contexto), *entrenamiento/adaptación* y *propagación*. A cada uno de estos se los independizó de los otros, teniendo en cuenta que muchos de los modelos vistos comparten algunos de ellos y varían en otros.

Por dar un ejemplo, tanto una red *backpropagation*, como el *ANFIS* comparten el mismo algoritmo de *propagación* (algoritmo *feedforward*). El propio algoritmo de entrenamiento del *ANFIS* puede también verse como una variante del *Backpropagation*, pero la forma de inicializarse es completamente distinta entre ellos. *Kohonen* también puede ser propagado mediante el algoritmo *feedforward* y su entrenamiento es totalmente distinto, y sin embargo el algoritmo *Nguyen-Widrow* de *inicialización* puede ser usado tanto en *Backpropagation* como en *Kohonen*.

Además de distinguir las operaciones principales como diferentes variabilidades, también se separó cada una de ellas en *algoritmo general/global* y *algoritmo local*. El primero se refiere a aquel que controla el proceso a nivel de la red en su globalidad (por ejemplo presentación de datos de entrada, propagación, comparación con salida de referencia, orden de adaptación, etc.) mientras que el segundo es efectuado localmente por cada *unidad de procesamiento*. Por ejemplo, la propagación global para una red feedforward sería "presentar entrada a la capa de entrada, propagar hacia las capas superiores, obtener salida en capa de salida". Localmente, cada unidad de procesamiento procesa la operación *propagar* como "recibir entrada, calcular valor de función de red, calcular activación, calcular salida, publicar salida".

Para las redes cuyos algoritmos son supervisados se identificó como variabilidad a la *función de costo* utilizada para evaluar el desempeño de la red y tomar acciones correctivas (por ejemplo, en , *Backpropagation*). Si bien el ejemplo más común es el *error cuadrático medio*, la decisión tomada fue que no se debería fijar esto ya que quizá alguien desee utilizar algún otro tipo de función.

Por último, se identificaron en forma separada cada una de las funciones que intervienen en la propagación local de una unidad de procesamiento, es decir, las funciones de *red (o membrana)*, *activación* y *salida*. La primera es una función de $\mathbf{R}^{2N} \rightarrow \mathbf{R}$ para una unidad con N entradas con sus N señales de entrada y N pesos asociados. Las otras dos son modeladas como funciones $\mathbf{R} \rightarrow \mathbf{R}$, suponiendo que la *activación* puede no ser algebraica (por ejemplo una ecuación de recurrencia o función algorítmica, etc.), pero no forzando este concepto por lo que queda libre al desarrollador.

5.4. Diseño

El principal énfasis en esta etapa fue puesto en facilitar y fomentar el desarrollo *ortogonal* de los distintos algoritmos que gobiernan el funcionamiento de una red neuronal artificial. Por *desarrollo ortogonal* se identifica a aquel que permite variar en forma independiente una parte del funcionamiento del sistema sin afectar al resto en lo que refiere a implementación e interfaz.

En particular, para una red neuronal artificial, fueron identificadas una serie de operaciones fundamentales comunes a todo sistema de este tipo en significado y cuyas distintas implementaciones caracterizan a cada tipo de red. Estas operaciones son *inicialización*, *calibración*, *propagación*, *entrenamiento* y *adaptación*.

En el diseño esto se traduce en una clase `NeuralNet` concreta cuya funcionalidad es extendida mediante *manejadores* (denominados *managers* en el framework) para cada tipo de operación. El *manejador de*

adaptación se denomina `AdaptationManager`, el de *propagación* `PropagationManager`, y el de *calibración* `CalibrationManager`.

El diseño obtenido lleva la ortogonalidad un poco más al extremo, permitiendo la variación independiente de los aspectos *globales* y *locales* de un algoritmo. Todos estos conceptos serán detallados en las siguientes subsecciones.

5.4.1. Clases Principales

La principal clase es `NeuralNet`. Esta no es una clase heredable sino que las distintas implementaciones de redes neuronales deben definir *managers* adecuados para lograr el efecto deseado. Esto hace que `NeuralNet` no sea un *hotspot*. En todo caso, `NeuralNet` puede ser vista como una *fachada* para todo el *subframework*, ya que define el comportamiento general de una red neuronal artificial a través de su interfaz.

Además, `NeuralNet` almacena la estructura y estado de la red en la forma de *unidades de procesamiento* representadas por la clase `NetNeuron` y conexiones entre unidades, modeladas por la clase `NetConnection`.

La clase `NetNeuron` es la segunda más importante, además de ser uno de los *hotspots* del subframework. Esta clase define el comportamiento básico de una unidad de procesamiento y provee una interfaz para la *propagación*, *calibración* y *adaptación* local de la unidad. Esta no es una clase *abstracta* ya que define un comportamiento concreto básico que consta de un algoritmo general de *propagación* de las señales, otro de calibración básica y una implementacion *no operante* para la *adaptación*.

En lo que respecta al modelo *computacional* de cada unidad, se define un mecanismo mediante el cual cada unidad puede ser configurada con su propia *función de red*, *función de activación* y/o *función de salida* en forma independiente mediante instancias de las clases `NetFunction` y `UnitFunction`.

`NetFunction` define la forma general para las *funciones de red*, es decir, funciones de $\mathbf{R}^N \rightarrow \mathbf{R}$, siendo N la cantidad de entradas de una unidad particular. Estas funciones toman además un conjunto de N parámetros correspondientes a los N pesos de las conexiones de entrada.

`UnitFunction` defune funciones de $\mathbf{R} \rightarrow \mathbf{R}$ y es utilizada para modelar las distintas funciones de *activación* y *salida* posibles de las unidades.

`NetConnection` no es mas que un enlace entre dos unidades de procesamiento con un *peso* asociado que indica la *intensidad* con que la salida de la unidad de origen afecta a la entrada de la de destino. La única particularidad de esta implementación es que admite un *learning rate* propio y la posibilidad de definir el *peso* como parámetro *interno* a la instancia o *externo*.

5.4.2. Hotspots

Hasta ahora no se ha aclarado el significado de las operaciones anteriormente nombradas: *inicialización*, *calibración*, *propagación*, *entrenamiento* y *adaptación*. Esto se debe a su correspondencia directa con los hotspots del subframework que serán detallados en este apartado. La posible excepción la hace la *inicialización*, que refiere a una cuestión *estructural* del sistema y que cada algoritmo particular define de acuerdo a su estructura. La necesidad de una operación como la *inicialización* depende de la posibilidad de tener diversas estructuras de unidades y conexiones coexistiendo con diversos algoritmos para lograr el funcionamiento de un sistema complejo como lo es la red neuronal artificial.

La primera operación fundamental es la *propagación de las señales* dentro de una red. Una red neuronal artificial recibe siempre una entrada en forma de vector numérico y la propaga dentro de su estructura de alguna forma hasta obtener una salida *estable*. Para distintos tipos de red la forma en que esto se lleva a cabo varía mucho. Por ejemplo, una red tipo *feed forward* toma la entrada en su *capa de entrada* y la propaga según un *orden topológico* hacia su capa de salida. Por otro lado, una red *recurrente* como *Hopfield* requiere de varias iteraciones para producir una salida desde que se carga una entrada en sus unidades.

Continuando con el ejemplo de *Hopfield* existen variantes *síncronas* y *asíncronas* para el mecanismo de propagación de dichas redes recurrentes. La variante *síncrona* implica el disparo simultáneo de todas las unidades de la red en base a un mismo estado previo, mientras que en el modelo *asíncrono* las unidades disparan en forma aleatoria una por vez y sin importar el orden del sistema.

Todos estos comportamientos pueden ser definidos implementando especializaciones de `PropagationManager`. El framework, para facilitar la tarea de estas implementaciones en lo que refiere al caso de *modelo de sincronismo* anteriormente mencionado, define un modelo de propagación de *unidades de procesamiento* en dos etapas: *propagación* y *disparo*.

La *propagación*, a nivel de *unidad de procesamiento* implica calcular una *salida potencial* de la unidad en base a su estado interno y entrada actuales. Esta salida no es *disparada* mientras no se le indique a la unidad mediante un método especial `fire()`.

De esta manera las distintas implementaciones de este *manager* pueden administrar el orden de propagación y disparo de las unidades para lograr el efecto deseado en lo que refiere a sincronía.

La *calibración* se refiere a la obtención de un estado propicio para lograr una rápida y efectiva adaptación durante el entrenamiento de la red, en base o no a un conjunto de vectores de calibración especiales. En general esto se traducirá a descubrir los *rangos de variación* de las distintas entradas y ajustar los parámetros iniciales del sistema para situarlo en una posición que minimice la probabilidad de caer en mínimos locales o generar *unidades muertas* durante el entrenamiento.

Por último el *entrenamiento* y *la adaptación* son ambas operaciones administradas a nivel global por `AdaptationManager`, dejando los detalles de nivel local a cada unidad para las distintas implementaciones de `NetNeuron`. Esto permite *ortogonalizar* el desarrollo de algoritmos generales como *backpropagation* (lógica general del algoritmo, secuencia de adaptación, etc.) y sus variantes a nivel global, de sus variaciones a nivel local (es decir, en la adaptación de cada unidad). Una implementación de este esquema está presente en el framework y es justamente la del algoritmo *backpropagation* por `BpnAdaptationManager` con dos variantes `BpnNeuron` y `QpNeuron` para *backpropagation con momentum* y *quickpropagation* (ver [16] por descripciones de estas variantes).

Mas aún, no solo se aumenta el reuso de código con este esquema sino que es posible definir redes *híbridas* que combinen unidades de distintas variantes de un mismo algoritmo, comandadas a su vez por un mismo algoritmo general, lo que agrega flexibilidad al sistema y lo hace atractivo para el prototipado de nuevos tipos de redes.

Por último, para el caso particular de redes con entrenamiento *supervisado* se define una subclase especial de `AdaptationManager`, `SupervisedAdaptationManager`, que a su vez utiliza una instancia de `ObjectiveFunction` para evaluar el *error* en la salida de la red respecto a una referencia dada, al propagar una entrada asociada a esa referencia. Como ejemplo, `BpnAdaptationManager` es una subclase de `SupervisedAdaptationManager` que utiliza la función `MseObjectiveFunction`[5] para evaluar el error en la salida de la red.

5.4.3. Funcionalidad adicional

Aparte de la funcionalidad adicional "heredada" del framework general, el subframework de redes neuronales artificiales define un mecanismo propio para el almacenamiento y carga de sus datos. En particular, fue definido un "filtro de exportación/importación" que puede ser utilizado para transferir redes neuronales tipo *backpropagation* desde y hacia el simulador de redes neuronales artificiales de la Universidad de Stuttgart, el SNNS [17].

5.5. Implementación

Los detalles de la implementación de este subframework se remiten a los criterios utilizados a nivel general en el framework. Sin embargo, cabe destacar el mayor énfasis puesto en el *desempeño* del sistema en tiempo de ejecución, dado que las redes neuronales artificiales suelen requerir un gran número de cálculos para su entrenamiento. El uso excesivo del *polimorfismo dinámico*, que suele ser tentador por lo elegante que resultan los diseños resultantes, fue evitado en la medida de lo posible debido a la degradación del desempeño que esto suele tener como consecuencia.

[5] MSE significa *Mean Square Error*.

5.6. Implementaciones disponibles para usar del subframework de redes neuronales

Se proveen implementaciones de administradores de entenamiento y neuronas para los algoritmos Backpropagation (con momentum), Quickpropagation, Kohonen SOM y ANFIS. Como función objetivo está disponible el *error cuadrático medio*.

Para la propagación se dispone de un algoritmo *feed forward* que descubre la topología automáticamente. Este algoritmo es adecuado para todas las redes implementadas nombradas anteriormente.

Para la *calibración* se implementó un algoritmo inspirado en el método de inicialización de pesos de Nguyen-Widrow [19], que minimiza la probabilidad de que se produzacan "neuronas muertas"durante el entrenamiento.

Por último se definieron funciones de *red* tipo *sumatoria*, *productoria*, *normalización de entradas*, *distancia*, y funciones de activación/salida tipo *sigmoide*, *inversa normalizada*, *modulo*, *gaussiana*, *campana (bell)* y *lineal*.

6. Aplicación de prueba

En esta parte del trabajo se plantea el diseño de una aplicación que ponga a prueba al framework de clasificación frente a un problema real. El problema planteado es un problema sencillo, ya que el objetivo aquí no se centra en las visiscitudes de la doctrina de reconocimiento de patrones en sí, sino en probar la funcionalidad del framework como herramienta de diseño.

El problema en cuestión es la *clasificación de maderas de eucalyptus* en base a un criterio de *similitud*. La idea es obtener un conjunto de unas pocas clases de madera de eucalyptus de forma de que en cada grupo las maderas sean lo suficientemente similares entre ellas en aspecto.

La necesidad de una clasificación de este tipo surge de la intención de introducir la madera de eucalyptus al mercado de la carpintería, especialmente en aquellas áreas en las cuales importa la uniformidad de color y aspecto de la madera utilizada en un mismo producto.

Para esta aplicación los requerimientos previstos implican la adquisición de las imágenes de las maderas en tiempo real mediante una cámara, y la comunicación de los resultados de clasificación a un sistema SCADA[6] para un eventual control de un dispositivo de accionamiento que permita dirigir los listones de madera hacia uno de los hipotéticos canastos.

6.1. Diseño del sistema de clasificación

Dado que el problema planteado se basa en la percepción subjetiva del ojo humano del color de las maderas, las características elegidas para conformar los patrones fueron basadas en las componentes HSV de las imágenes obtenidas de la superficie de las maderas.

La primera decisión importante fue entonces utilizar como patrón el vector *HSV promedio* de la imagen. Esta decisión surge de las propiedades del modelo de color *HSV* [2], que pretende modelar los parámetros más relevantes para el ojo humano en la detección del color[7]. La elección del *promedio* HSV también deriva del hecho de que el ojo humano *promedia* la imagen cuando la resolución natural no le alcanza para percibir los detalles, en este caso, la veta de la madera a partir de cierta distancia.

Ya que no se trata aquí de ubicar a las maderas en un conjunto de clases predefinidas (y estandarizadas) sino que la agrupación obtenida puede ser arbitraria, siempre y cuando se cumpla el requisito básico de *similitud* entre maderas de la misma clase, se optó por un algoritmo de clasificación *no supervisado*.

El algoritmo en cuestión es el *K-Means*, debido a su sencilla implementación y aparente idoneidad para el problema a resolver. El *K-Means* es un algoritmo de *agrupación no supervisado* que toma como único parámetro a la cantidad de grupos a crear: K. El valor óptimo de K es el mínimo necesario para lograr el cumplimiento del requisito de similitud entre maderas del mismo grupo, debido a que un mayor número de

[6] Supervisory Control And Data Acquisition.

[7] Tener en cuenta que la relación de similitud buscada dentro de los grupos formados es respecto al ojo humano.

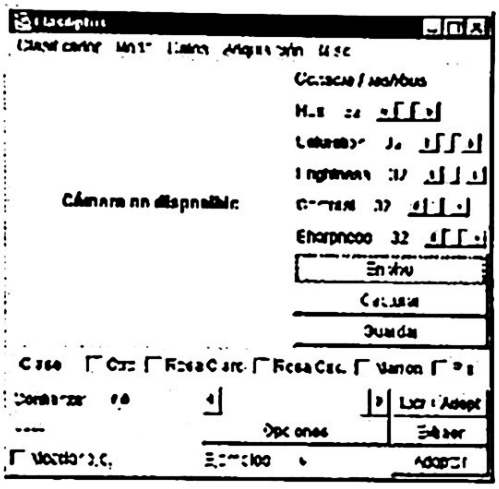

Figura 1: Interfaz de usuario

grupos aumenta los costos de administración del stock (derivado de un mayor número de compartimientos, más categorías y menor disponibilidad de madera en cada categoría).

6.2. Implementación

La aplicación de prueba fue implementada con el *framework* como base para toda la parte de clasificación, mas módulos para adquisición y preprocesamiento (no abarcados por el framework en su estado actual) y otros tantos para la comunicación con el SCADA utilizado para "publicar" los resultados.

El resultado final permite la creación de varios tipos de sistemas de clasificación (todos los implementados como parte integral del framework), a saber, *Backpropagation*, *Bayes* y *K-Means*. De esta forma pudo ser utilizada como plataforma de prueba de los algoritmos supervisados implementados como parte del framework, utilizando K-Means para definir las clases y producir en base a ellas un conjunto de entrenamiento supervisado.

En las figuras 1 y 2 puede verse una muestra del aspecto que tiene la interfaz gráfica diseñada. Esta interfaz permite crear, configurar, calibrar, entrenar y almacenar un clasificador dado. También permite crear, cargar y almacenar ejemplos de entrenamiento, seleccionar el origen de las imágenes (cámara archivo), seleccionar el método de calibración de la cámara, ajustar los parámetros de adquisición (brillo, contraste, saturación, etc.).

7. Resultados

7.1. El Framework como herramienta de desarrollo

Algo que vale la pena destacar es la naturalidad con que los conceptos y las interfaces definidas fueron utilizadas a la hora de desarrollar aplicaciones concretas. Cabe notar que bajo el mismo esquema implementaron algoritmos supervisados y no supervisados, entrenados "por lotes" (*batch*) o "en línea" (*online*), paramétricos o no. Por supuesto, esta naturalidad no fue conseguida sin antes pasar por más de un ciclo de desarrollo, desde el planteo del trabajo. La posible excepción, por no haber sido probada aun, es la implementación de algoritmos no estadísticos sino *estructurales* o KBS, por nombrar algunos paradigmas.

También puede nombrarse el gran aporte que el sistema de persistencia basado en XML hizo a aplicación. El almacenamiento de clasificadores y ejemplos probó ser de mucha utilidad, así como lo fue

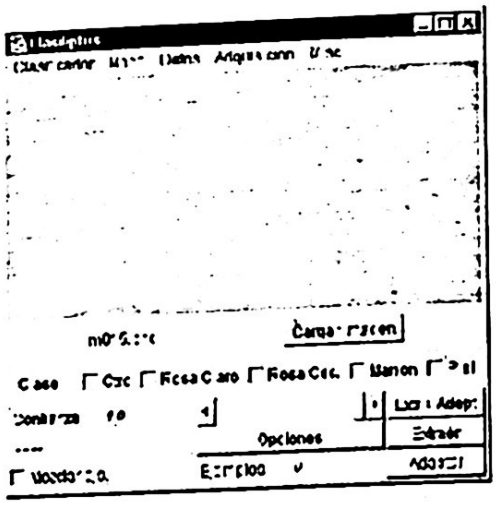

Figura 2: Interfaz de usuario

el poder modificar o generar dichos archivos manualmente o desde otras aplicaciones y cargarlos desde el programa para algunos casos especiales.

Otra de las ventajas del framework es su interfaz genérica para la configuración, entrenamiento, clasificación y evaluación de desempeño, permitiendo con la misma lógica utilizar todos los clasificadores soportados/implementados. La *matriz de confusión* provee, para sistemas supervisados, una buena codificación que sintetiza el comportamiento del sistema para una aplicación general. En particular para la *configuración* mostró ser de mucha utilidad la interfaz *Configurable* definida para tales propósitos en el framework.

7.2. La clasificación de maderas

El algoritmo utilizado finalmente fue entonces el *K-Means*, obteniéndose resultados para diversas cantidades de *grupos* o *clusters*.

Este algoritmo, de muy sencilla implementación, probó ser una opción suficiente para los requerimientos impuestos sobre el problema y dentro de las opciones que se tenían para elegir.

No se descartan otras alternativas de algoritmos no supervisados como *K-nearest-neighbors* o *Kohonen*. K-Means fue una opción lógica por su sencillez de implementación y depuración.

En todo caso se utilizó una *métrica* para el *espacio de características* que ponderó de igual forma a las tres componentes H, S y V, no habiéndose realizado análisis específicos respecto a la sensibilidad relativa que cada componente tiene a la vista humana.

De todas formas, el propio formato *HSV* está definido en base a estos criterios [20], lo que ayudó sin duda a obtener un resultado aceptable con un algoritmo fuertemente dependiente de la métrica utilizada como el K-Means.

En la figura 3 pueden verse cuatro ejemplares correspondientes a cada uno de los cuatro *grupos* definidos por el clasificador *K-Means* con $K = 4$.

Se observó que una separación de 4 o 5 grupos de maderas resulta en grupos cuyas maderas son aceptablemente similares, al menos según los criterios de los *expertos* que observaron los resultados. Cabe notar entre estos expertos se encuentra al menos un especialista en el tema de las maderas: el Ing. Roberto Bavosi de la empresa *Eufores*, y un especialista en tratamiento de imágenes: el Dr. Guillermo Sapiro, profesor de la Universidad de Minnesota, EE.UU. .

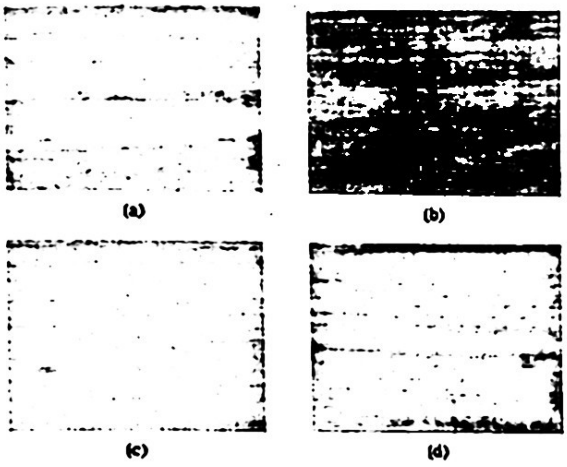

Figura 3: Ejemplos de maderas pertenecientes a los cuatro grupos creados.

7.3. Bayes y Backpropagation

El otro punto pendiente es la prueba de los algoritmos de clasificación implementados. Para probar dichos algoritmos, en particular *Backpropagation* y *Bayes*, se utilizó como *experto* al propio algoritmo *K-Means*.

Con la agrupación realizada por K-Means se crearon ejemplos de entrenamiento supervisado y con ellos fueron entrenados y probados los algoritmos Backpropagation y Bayes. Las pruebas fueron realizadas para diversas cantidades de grupos ($K = 2,3,4,5,6,7,8,9,10$). Los resultados mostrados a continuación corresponden a $K = 4$, cantidad que fue encontrada como las más adecuada para el problema.

Los clasificadores basados en *Backpropagation* y *Bayes* fueron entrenados con la mitad de los ejemplos generados (que suman 282 ejemplos) y probados con la otra mitad, observándose los resultados de las matrices de confusión obtenidas. A continuación se listan tablas y figuras que muestran los resultados obtenidos.

44	7	0	6	0.77
0	97	0	0	1.00
0	2	47	0	0.96
2	4	0	70	0.92
0.96	0.88	1.00	0.92	0.92

Cuadro 1: Matriz de confusión para una red Backpropagation para K=4. Los parámetros utilizados: $epocas = 500$, $neuronasocultas = 25$, $learning_rate = 0{,}001$, $momentum = 0{,}8$.

54	2	0	1	0.95
1	90	6	0	0.93
0	1	48	0	0.98
1	0	0	75	0.99
0.98	0.97	0.89	0.99	0.96

Cuadro 2: Clasificador de Bayes para K=4

8. Conclusiones

Obtuvimos como resultado de este trabajo un framework de clasificación de patrones que en su primera versión permite la creación de aplicaciones en forma razonablemente rápida y con un buen nivel de flexibilidad. Esto se cumple especialmente dentro de la familia de clasificadores estadísticos y con algún pequeño inconveniente, pero no impedimento, para los otros dos tipos más conocidos de clasificadores: KBS y estructurales, como puede ser la utilización de elementos de tipo *double* para representar etiquetas o símbolos.

La posible excepción son quizá los clasificadores lógico-combinatorios, cuya estructura en algunos casos puede requerir tipos especiales no previstos actualmente.

En cuanto al paquete como herramienta de software se pueden destacar:

- En sí, el paquete es extensible, pudiéndose mejorar su infraestructura para, por ejemplo, abarcar paradigmas como el lógico-combinatorio.
- Está ampliamente documentado, a través de su manual de usuario y su documentación de API (interfaz de programación).
- Es suficientemente genérico como para abarcar una buena parte de los algoritmos de clasificación existentes.
- La interfaz de programación resulta suficientemente intuitiva, permitiendo un rápido desarrollo de nuevos prototipos de clasificadores.
- La arquitectura del paquete no agrega recargas al tiempo de ejecución.

Entre las decisiones más acertadas está sin duda la elección de los estándares *DOM* y *XML* para la codificación, almacenamiento y transmisión de los datos que definen a un sistema de clasificación, su configuración y estado. Esto probó ser de enorme utilidad en la aplicación final, minimizando la lógica de entrada/salida para prácticamente cualquier operación de este tipo para patrones, ejemplos, clasificadores, resultados, etc.

Otra decisión acertada fue sin duda la incorporación de un mecanismo genérico de *configuración*, facilitando enormemente el desarrollo de la aplicación final.

El mecanismo de control de *nivel de logging o verborragia* también ayuda a monitorear los algoritmos implementados con detalle o reducir los mensajes al mínimo según se desee o necesite.

8.1. Subframework de redes neuronales

Este módulo terminó ocupando gran parte del paquete total, quizá debido a la mayor especificidad de los algoritmos involucrados. El resultado es una base para el desarrollo de redes neuronales que bien puede utilizarse aparte del framework general para otros fines que no sean clasificación.

El diseño obtenido brinda un buen nivel de ortogonalidad en base a la formulación de una red neuronal como una clase compleja, formada por unidades, conexiones y *managers* que controlan cada aspecto de su funcionalidad.

La estrategia de dividir los algoritmos en una parte *global* (dentro de los managers) y *local* (dentro de las neuronas) permite definir redes neuronales *híbridas* que utilicen distintas variantes de un mismo algoritmo (por ejemplo *Backpropagation*) en neuronas distintas (ver [16]). Creemos que esto es un punto fuerte del framework que permitirá, para quien lo utilice, experimentar con nuevos tipos de redes neuronales en base a este paradigma.

Una posible limitación es la no utilización de *managers* a nivel *local* (dentro de una neurona) como se hizo en `NeuralNet` para los algoritmos globales.

Otra limitación es la falta de un mecanismo de adaptación *general* para *todos* los parámetros de una unidad, no sólo los *pesos de las conexiones*. Si bien para la gran mayoría de las redes existentes esto no es problema, sistemas como el *ANFIS* [10] necesitan esta cualidad para su correcta adaptación. De hecho,

la actual implementación del ANFIS *no* modifica uno de los parámetros importantes debido a este diseño (ver [21] por detalles).

El problema de la falta de un mecanismo *amigable* de construcción de sistemas de clasificación se nota fuertemente en este módulo. Si bien por esta razón se creó la clase `NetBuilder` desde sus comienzos, su diseño no ha llegado a una forma aceptable aún y su uso es un tanto engorroso.

Por último, de la experiencia en el *uso* de las redes neuronales se concluye que sería muy útil tener algún mecanismo de *reportes* o *auditoría* de la evolución interna de los parámetros y las señales durante el entrenamiento. Se pensó en un mecanismo de *auditoría*, implementable mediante otro *manager* (por ejemplo "NetAuditor"), pero no se pudo llegar a implementar dicho mecanismo dentro del tiempo admisible.

8.2. El framework como base para la aplicación final

Del uso del framework en la aplicación es que surgieron la mayoría de las conclusiones respecto a él citadas anteriormente. En general, una buena prueba de la utilidad del framework es el tiempo relativo de desarrollo de la aplicación que tomó la parte de clasificación.

La aplicación comenzó a ser desarrollada a mediados de Enero del 2002 y fue culminada en la primera semana de Abril del mismo año. De los casi 3 meses que llevó el desarrollo y puesta a punto, solo una semana de implementación y depuración estuvo directamente ligada a la parte de clasificación. El resto del tiempo fue dedicado a los módulos de *adquisición*, *preprocesamiento y calibración* y *comunicación con el SCADA* y a la interfaz gráfica (junto con la interfaz de comunicación Java/C++: JNI).

9. Conclusiones finales

Finalmente se obtuvo un framework que en su primera versión permite desarrollar sistemas de clasificación de patrones con relativa facilidad, a la vez resolviendo algunos problemas comunes a los que un desarrollador se enfrenta al crear este tipo de aplicaciones.

9.1. Ventajas de utilizar el framework

Al tener resueltos la estructura lógica y el diseño general del sistema de clasificación, el desarrollador puede concentrar sus esfuerzos en el perfeccionamiento de las etapas de extracción de características clasificación.

Además se ahorrará tiempo en resolver problemas comunes como la codificación interna, el almacenamiento y la recuperación de la información almacenada por un sistema arbitrario, tanto de entrenamiento como de configuración.

El paquete está desarrollado en C++, lenguaje muy difundido en el área de la computación científica y que combina eficiencia con un alto nivel de abstracción. El desarrollador podrá beneficiarse de estas características al trabajar sobre este framework.

9.2. Desventajas

El grado de generalidad alcanzado por en Framework tiene como consecuencia negativa el inevitable enlentecimiento general del sistema debido a la lógica adicional requerida para manejar cierta diversidad de situaciones. Parte de esta lógica está implícita en el mecanismo de *polimorfismo dinámico*, que forma parte importante del paradigma de Orientación a Objetos bajo su implementación en el lenguaje C++. Claramente, una aplicación realizada desde cero para una aplicación puntual será seguramente más eficiente en tiempos de ejecucion que una realizada para el Framework con idéntica funcionalidad.

También habrá un tiempo de adaptación por parte del desarrollador, necesario para que éste pueda captar la filosofía general que gobierna el diseño del framework y así implementar correctamente aplicación en base a este último.

10. Trabajo Futuro

Quedan pendientes todas las posibles mejoras mencionadas: facilitar la construcción de sistemas de clasificación, extender el framework para comprender a *todo el sistema de clasificación* (es decir, agregando *adquisición* y *extracción*).

También sería muy interesante definir una interfaz *asíncrona* para el sistema, para así permitir un proceso en *pipeline* en el cual cada etapa trabaje continuamente y la comunicación entre etapas sea mediante *eventos*. Esta interfaz requeriría el uso de técnicas de *programación concurrente/multiproceso* y *señalización* e implicaría un importante tiempo de desarrollo, con impacto global en todo el framework. Cabe notar que este objetivo fue tenido en mente desde el comienzo del desarrollo, pero no hubo lugar para su realización dentro del tiempo disponible.

Por la parte de las redes neuronales, es necesario focalizar en aumentar la velocidad de cálculo. Si bien la velocidad actual no es mala, el entrenamiento de redes neuronales suele ser largo y costoso en tiempos de ejecución. Varias alternativas de diseño, de carácter demasiado técnico para incluir aquí, quedan planteadas como posibles mejoras.

Siguiendo con las redes neuronales, un importante avance en el diseño es la generalización del mecanismo de adaptación para poder modificar *todos* los parámetros de una unidad y no solo los pesos de las conexiones, con vistas a mejorar las implementaciones de redes como el ANFIS nombradas anteriormente.

Referencias

[1] R. O. Duda and P. E. Hart, *Pattern Classification And Scene Analysis.* Wiley/Interscience, 1974.

[2] J. T. Tou and R. C. Gonzalez, *Pattern Recognition Principles*, ch. 4. Addison-Wesley, 1974.

[3] J. R. Shulcloper, A. G. Arenas, and J. F. M. Trinidad, *Enfoque Lógico Combinatorio al Reconocimiento de Patrones.* Instituto Politécnico Nacional - Mexico, 1999.

[4] F. J. Cortijo, "Reconocimiento de patrones y análisis de imágenes," tech. rep., Universidad de Granada, 1998.

[5] R. C. Crida, *A Machine Vision Approach to Rock Fragmentation Analysis.* PhD thesis, University of Cape Town, 1995.

[6] A. S. Pandya and R. B. Macy, *Pattern Recognition with Neural Networks in C++.* CRC Press, 1996.

[7] R. O. Duda, *Pattern Recognition Tutorial - http://www-engr.sjsu.edu/~knapp/HCIRODPR/PR_model/PR_model.htm.*

[8] N. Chomsky, "Three models for the description of language," in *Proc. of the Group of Information Theory*, vol. 2, pp. 113–124, 1956.

[9] M. Stefik, *Introduction to Knowledge Systems.* Morgan Kaufmann, 1995.

[10] J.-S. R. Jang, *Neuro-Fuzzy Modeling: Architectures, Analyses and Applications.* PhD thesis, University of California, 1992.

[11] W. W. W. Consortium, *DOM Reference - http://www.w3.org/TR/REC-DOM-Level-1/.*

[12] W. W. W. Consortium, *XML Reference - http://www.w3.org/TR/REC-xml1/.*

[13] T. A. X. Project, *Xerces C++ XML Parser - http://xml.apache.org/xerces-c/.* The Apache Group.

[14] A. Lumsdaine, J. Siek, and L.-Q. Lee, *The Matrix Template Library http://www.osl.iu.edu/research/mtl/.* Open Systems Laboratory - Indiana University.

[15] D. M. S. James A. Freeman, *Neural Networks: Algorithms, Applications, and Programming Techniques.* Addison-Wesley, 1991.

[16] J. Gibb, "Back propagation family album," tech. rep., Department of Computing - Macquarie University, August 1996.

[17] University of Stuttgart - University of Tübingen, *SNNS User Manual, Version 4.2 - http://www-ra.informatik.uni-tuebingen.de/downloads/SNNS/SNNSv4.2.Manual.pdf.*

[18] R. Hech-Nielsen, *Neurocomputing.* Addison-Wesley, 1990.

[19] Matlab(TM), *Neural Network Tookbox Documentation - http://www.mathworks.com/access/helpdesk/help/toolbox/nnet/* MatLab(TM).

[20] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Addison-Wesley, 1992.

[21] I. Ramírez and A. Alcarraz, *Manual del Usuario de ClasiLiptus - http://www.iie.edu.uy/~nacho/clasiliptus/manual.pdf.* Instituto de Ingeniería Eléctrica - Facultad de Ingeniería, Julio Herrera y Reissig 565, Montevideo (Uruguay), 1997.