RAZONAMIENTO APROXIMADO. CUADERNILLO DE PRÁCTICAS

1.PREPARACIÓN DEL TRABAJO

- Descargar Weka: http://www.cs.waikato.ac.nz/ml/weka/downloading.html. No recomiendo descargar la versión de desarrolladores, al no ser estable. Haz la descarga para 64 bits si tu máquina está preparada para ello.
- Esta práctica está diseñada para Weka 3.6.12
- Descargar Eclipse.

2.TUTORIAL BÁSICO DE WEKA

2.1.LISTADO DE PAQUETES

- ***weka.core***: Paquete con las clases e interfaces que conforman la infraestructura de WEKA. Son comunes a los distintos algoritmos implementados en WEKA.
  - Define las estructuras de datos que contienen los datos a manejar por los algoritmos de aprendizaje
    - *Instances*: Clase que encapsula un *dataset* de datos junto con los métodos para manejarlo (e.j. creación y copia, división en subdatasets [entrenamiento y prueba] etc.).
    - *Attribute*: Clase que encapsula los atributos que definen un *dataset*
    - *Instance*: Clase que encapsula cada uno de los ejemplos individuales que forman un *dataset*, almacenando los valores de los respectivos atributos.
  - ***weka.core.neighboursearch***: subpaquete con la implementación de algoritmos y estructuras de datos para la búsqueda eficiente de vecinos.
- ***weka.classifiers***: Paquete con las implementaciones de algoritmos de clasificación (métodos de clasificación discreta como de predicción numérica).
  - Subpaquetes: *weka.classifiers.rules*, *weka.classifiers.lazy*, *weka.classifiers.trees*, *weka.classifiers.meta*, etc.
  - *Classifier*: Clase abstracta con métodos comunes a todos los clasificadores. Todos los clasificadores heredan de ella.
  - *weka.classifiers.evaluation:* subpaquete con funcionalidad para evaluar algoritmos de clasificación.
- ***weka.clusterers***: Paquete con las implementaciones de algoritmos de *clustering*.
  - *AbstractClusterer*: Clase abstracta con métodos comunes a todos los algoritmos de *clustering.*
  - *ClusterEvaluation*: Clase evaluadora de *clusters*.

2.2.CLASES BÁSICAS: DATASETS, ATRIBUTOS E INSTANCIAS

CLASE INSTANCES

Representación en memoria de una colección de ejemplos (*dataset*) descrito por un conjunto de atributos (*Attribute*). Contiene un conjunto de instancias/ejemplos (*Instance*) que almacenan conteniendo los valores de sus atributos.

Opcionalmente (ej. En clasificación) uno de los atributos podrá estar marcado como ***atributo de clase***. Esto habrá que tenerlo en cuenta al ejecutar algoritmos de clasificación. Habrá que configurar en el *main* cuál es el atributo de clase.

Métodos

- Constructores:
  - *Instances(java.io.Reader reader)*: Crea un *dataset* y lo carga desde el fichero ARFF al que apunta el *Reader*.
  - *Instances(Instances dataset)*: Crea un *dataset* copiando las instancias del *dataset* que se pasa como parámetro.
  - Instances(Instances dataset, int capacity): Crea un *dataset* vacío con la estructura del *dataset* que se pasa como parámetro.
- Manejar atributos
  - *void setClassIndex(int classIndex):* Establece el atributo de clase (valor en [0, numAttributes-1]).
- Manejar instancias
  - Instance instance(int index)**:** Recupera la instancia index-ésima.
  - Instance remove(int index): Elimina la instancia index-esima.
- Estadísticas: *numInstances()*,*numAttributes()***,** *num.Classes()*.

CLASE INSTANCE

Almacena un ejemplo o patrón (instancia). Internamente los valores de los atributos de cada instancia se representan como un vector de números reales (*double[]*), independientemente del tipo de los atributos.

Métodos

- *double classValue()*: devuelve el valor almacenado en el atributo clase en formato interno (es el índice de la etiqueta de la clase).
- *double value(int index)*: devuelve el valor de un atributo numérico (o el índice del vector del valor en los nominales (ej. classValue).
- void setValue(int index, double value): establece un valor determinado para un atributo.

CLASE LINEARNNSEARCH

Implementa la búsqueda de los vecinos más cercanos por fuerza bruta.

Métodos

- LinearNNSearch(Instances insts). Constructor con un conjunto de instancias.
- void setInstances(Instances insts). Establece las instancias que conformarán el vecindario (*neighbourhood*).
- setSkipIdentical(boolean skip). Establece la propiedad que incluye (o no) dentro de los vecinos devueltos aquellas instancias idénticas a la instancia objetivo (aquellas con distancia cero a la instancia).
- Instances kNearestNeighbours(Instance target, int kNN). Devuelve un nuevo *dataset* con los kNN vecinos de la instancia objetivo.
    - Puede devolver más de k vecinos en caso de que haya empates de distancia.
    - Los vecinos los devuelve ordenados por distancia.
- Instance nearestNeighbour(Instance target) Devuelve el vecino más cercano a la instancia objetivo en el vecindario.
- double[] getDistances(). Devuelve las distancias a los k vecinos más cercanos. Requiere haber llamado previamente a kNearestNeighbours

CLASE LINEARNNESEARCH

Hereda de LinearNNSearch e implementa un método adicional.

Métodos

- public int[] kNearestNeighboursIndices(Instance target, int kNN). Devuelve un vector con los índices de los vecinos más cercanos.

PRACTICA 2. DESARROLLAR UN CLASIFICADOR FUZZY KNN EN WEKA

3.CONSTRUCCIÓN DE UN CLASIFICADOR EN WEKA

Todos los algoritmos de clasificación heredan de **weka.classifiers.Classifier** y deben de implementar los siguientes métodos básicos:

- **void buildClassifier(Instances data)**: entrena el clasificador con el conjunto de entrenamiento (**Instances**) indicado
- **double classifyInstance(Instance instance)**: clasifica la instancia que recibe como parámetro. Exige haber invocado antes a **buildClassifier()**.
    - La estructura de la instancia (número y tipo de atributos) debe coincidir con la del objeto **Instances** usado en el entrenamiento
    - El valor devuelto (de tipo *double*) indica la clase predicha. Se corresponde con el índice de su etiqueta en el **FastVector** asociado al atributo clase.

- **double[] distributionForInstance(Instance instance)**: clasifica la instancia y devuelve un vector **double[]** con un componente para cada valor del atributo clase que cuantifica su probabilidad o importancia relativa (dependiendo del método de clasificación). Exige haber invocado antes a **buildClassifier()**.

```
/*Clasificador que devuelve siempre la clase mas probable en el dataset*/
public class MasProbable extends Classifier {
	private static final long serialVersionUID = 1L;
	double frecuencias[];// Vector con la frecuencia de cada clase
	public void buildClassifier(Instances data) throws Exception{
		frecuencias = new double[data.numClasses()];
		 //Contabilizar las frecuencias de cada clase
		 for (int i = 0; i < data.numInstances(); i++) {
			 frecuencias[(int) data.instance(i).classValue()]++;
		 }
		 for (int i=0; i<frecuencias.length; i++)
			 frecuencias[i]/= data.numInstances();
	 }
	 public double classifyInstance(Instance instancia) {
		 return Utils.maxIndex(frecuencias); }
	 public double[] distributionForInstance(Instance instancia) {
		 return Arrays.copyOf(frecuencias, frecuencias.length);}
 }
```

Para probar la clase haremos una clase Ejecutar que tenga un *main*:

```
import java.util.Random;
import weka.core.Instances;
import weka.classifiers.Evaluation;
import weka.core.converters.ConverterUtils.DataSource;
public class Ejecutar {
	public static void main(String[] args) throws Exception {
		Foo C;
		 C= new Foo();
		//Cargamos el dataset. Indicamos cuál es la clase objetivo
		DataSource source = new DataSource("data/iris.arff");
		Instances instances = source.getDataSet();
		instances.setClassIndex(instances.numAttributes() - 1);
		//Cross validation
		Evaluation eval = new Evaluation(instances);
		eval.crossValidateModel(C, instances, 5, new Random(1));
		System.out.println("RESULTADOS CLASIFICADOR FOO");
		System.out.println(eval.toSummaryString());
		System.out.println(eval.toClassDetailsString());
		System.out.println(eval.toMatrixString()); }
}
```

EJERCICIO 1

- **Crea un proyecto java que te permita ejecutar el ejemplo. Presta atención a los detalles.**
- **Este archivo se subirá en un .zip junto con el resto de archivos de la práctica.**
- **Los *datasets* de prueba los tienes disponibles en *Moodle***

4. CONSTRUIR UN KNNCRISP EN WEKA

Vamos a construir inicialmente un clasificador kNN básico, llamado CrispKNN. Aprovecharemos la clase **LinearNNSearch** para la búsqueda de vecinos y el cálculo de las distancias.

```
LinearNNSearch S = new LinearNNSearch(dataset);
S.setSkipIdentical(true);
Instances kNN =  S.kNearestNeighbours(instancia, k) ;
```

EJERCICIO 2

- **Descargate la plantilla CrispKNN y complétala.**
- **Añade al *main* del ejercicio anterior el código para ejecutar el algoritmo**

5.CONSTRUIR UN FKNN EN WEKA

EJERCICIO 3

- **Descárgate la plantilla FuzzyKNN y complétala.**
- **Añade al *main* del ejercicio anterior el código para ejecutar el algoritmo.**

6.CONSTRUIR UN K NEAREST PROTOTYPE EN WEKA

EJERCICIO 4

- **Descárgate la plantilla FuzzyNP y complétala.**
- **Añade al *main* del ejercicio anterior el código para ejecutar el algoritmo.**

PRACTICA 3. IMPLEMENTAR FUZZY CMEANS EN WEKA

3.CONSTRUCCIÓN DE UN ALGORITMO DE CLUSTERING EN WEKA

Todos los algoritmos de *clustering* implementan el interfaz *weka.clusterers.Clusterer* y deben de aportar los siguientes métodos básicos:

- **void buildClusterer(Instances data)**: calcula los *clusters* (grupos) para el dataset de entrenamiento indicado
- **int numberOfClusters()**: número de *clusters* resultantes (dependiendo del método concreto se especifica antes de entrenar o se calcula durante el entrenamiento)
- **int clusterInstance(Instance instance)**: indica el *cluster* al que pertenece la instancia pasada como argumento. [Exige haber invocado antes a **buildClusterer()**]
    - La estructura de la instancia (número y tipo de atributos) debe coincidir con la del objeto **Instances** usado en el entrenamiento