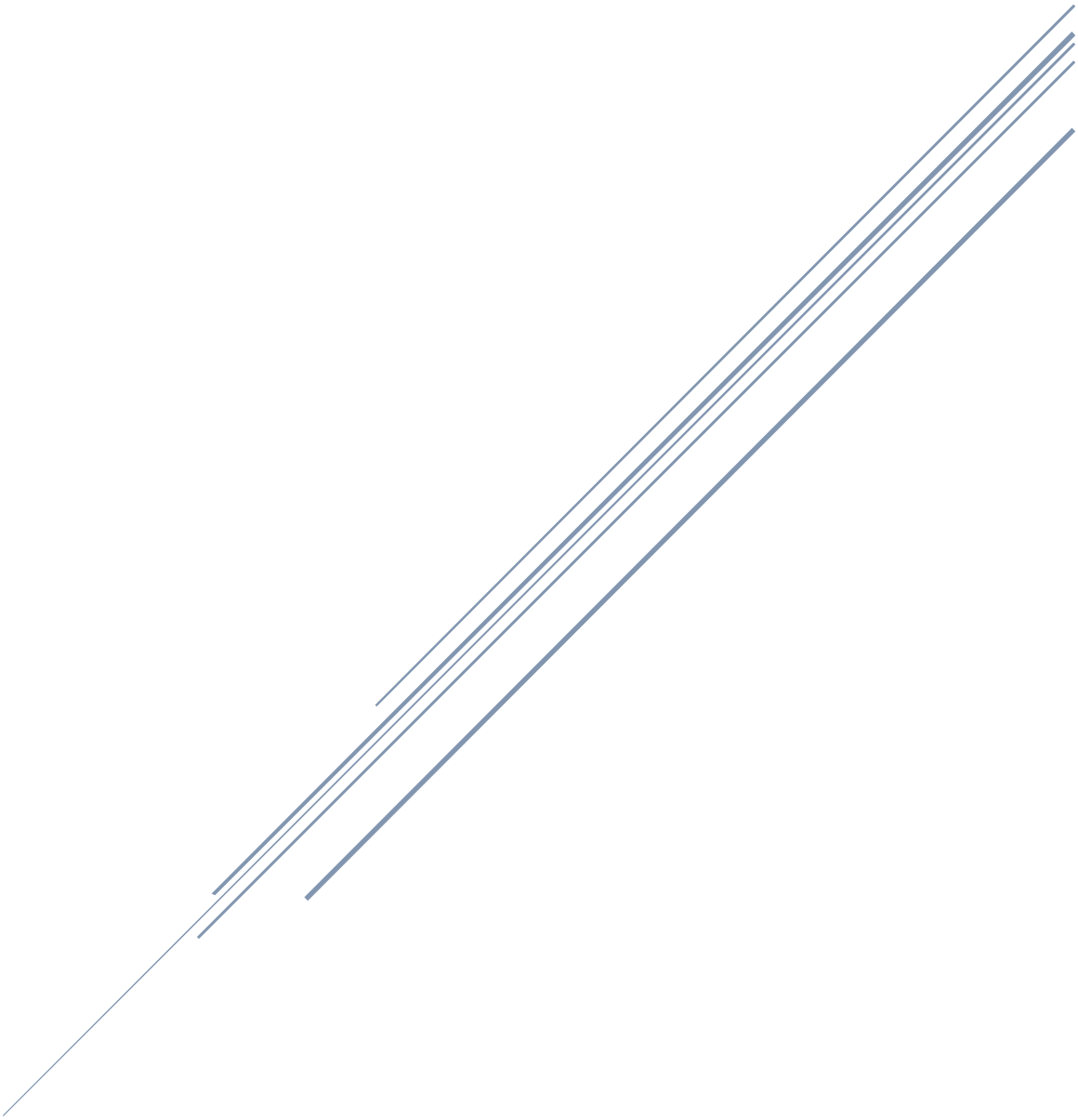
APRENDIZAJE AUTOMÁTICO

Práctica 1

Universidad Carlos III Madrid, Ingeniería Informática



IVAN AGUADO PERULERO – 100405871
JORGE SERRANO PÉREZ – 100405987

Contenido

1.- INTRODUCCIÓN

El objetivo de esta práctica es generar un modelo de clasificación y otro de regresión sobre el videojuego Pac-Man, para más tarde construir un agente automático empleando uno de los dos modelos. Para ello, primero se deberá generar un conjunto de ejemplos de entrenamiento que luego serán utilizados para generar ese modelo.

En este documento se divide en distintas fases. En la primera se recabará la información tanto de entrenamiento como de test para la clasificación y la regresión. En la segunda, se construirá el modelo de clasificación, en la tercera el de regresión, y en la cuarta el agente automático utilizando uno de los modelos anteriores. Por último, se responderán una serie de preguntas expuestas en el enunciado.

2.- FASE 1: RECOGIDA DE INFORMACIÓN

En esta primera fase se explicará la función de extracción programada y el mecanismo para incluir datos futuros en una instancia.

El objetivo es generar un archivo *.arff* que sea legible por Weka. Para ello, el primer paso es insertar la cabecera correspondiente de los archivos *.arff,* que consta de la etiqueta @RELATION, seguido de los distintos atributos de cada instancia (@ATTRIBUTE) y su tipo correspondiente, y finalmente los datos (@DATA). Esto se realiza en el fichero *game.py*, en nuestro caso al inicializar el juego, donde especificamos también el nombre del archivo que vamos a abrir (o crear en caso de que no exista) y el *FLAG* que indica que no hay que sobrescribir los datos, si no añadirlos al final del fichero (‘a’).

El siguiente paso es dividir los atributos de cada instancia correctamente para que se correspondan también con su tipo de dato (*numeric* o *nominal*). Por ejemplo, es necesario dividir las posiciones de cada fantasma en sus coordenadas x e y, para lo que hemos utilizado un bucle que recorre todas las posiciones y luego la función *join* para concatenarlas en una cadena. Todo ello está incluido en la función *printLineData* del fichero *bustersAgents.py*, que devuelve la variable msg en la que hemos ido agrupando todos los atributos separados por comas. Destacar que esta función debe codificarse tanto en el pacman controlado por teclado como el agente automático.

También se nos pedía incluir en el fichero la acción que se ha ejecutado durante el turno, que la podemos obtener llamando al método *chooseAction*. Este atributo será la clase que se clasificará en la fase 2 que veremos posteriormente.

Por otro lado, debíamos también codificar un mecanismo capaz de guardar en una misma línea (instancia) información del próximo tick: la puntuación siguiente (*nextScore*). Nuestro planteamiento para ello ha sido sencillo, cuando la distancia al pacman es igual a 1, quiere decir que el pacman está al lado de un fantasma y se lo comerá en el siguiente turno. Por ello, la puntuación ascenderá 200 puntos -1 que habrá que quitar del desplazamiento del siguiente

turno, es decir, la puntuación siguiente será la puntuación de ese momento más 199 en el caso de comerse un fantasma. Si se come una bola de comida, habrá que sumar 99 en vez de 199, pero el mecanismo es el mismo. Por último, si el pacman no está cerca ni de un fantasma ni de una bola de comida, entonces para saber la puntuación siguiente tan solo habrá que restarle uno a la actual, ya que es lo que se ha establecido que suceda cuando pasa el turno.

Finalmente, los atributos que hemos seleccionado han sido: *living pacman*, *living ghost* (*True* si vive, *False* si no, siendo el del pacman siempre *False*), *ghost position* (numeric) de cada fantasma, *ghost direction* (*North*, *South*, *East*, *West*, *Stop*), *ghost distance* (*numeric*), *dot distance* (*numeric*), *score* (*numeric*), *next score* (*numeric*), *pacman position* (*numeric*), *pacman direction* (*North*, *South*, *East*, *West*, *Stop*) y *pacman action* (*North*, *South*, *East*, *West*, *Stop*).

Para cerrar esta fase, debíamos crear con la función implementada los datos de entrenamiento y los de test que luego usaríamos en las fases posteriores. Estos datos se agrupan en seis ficheros distintos, tres de ellos con el agente manual y los otros tres con el automático. Estos a su vez se dividen en un fichero de entrenamiento con un número de instancias entre 300 y 500, y dos ficheros de test con 100-200 instancias, usando los mismos mapas que en el entrenamiento en uno de ellos y mapas diferentes en el otro.

3.- FASE 2: CLASIFICACIÓN

En la segunda fase se construirá un modelo de clasificación sobre la acción que deberá ejecutar el pacman. Se incluirá una justificación de los algoritmos y atributos seleccionados, el tratamiento de datos llevado a cabo y un análisis de los resultados producidos por cada algoritmo, así como la justificación de la elección final.

TRANSFORMACIÓN DE DATOS

En primer lugar, se han estudiado los datos obtenidos de tal manera que el conjunto se comporte de mejor forma a la hora del aprendizaje automático y sean todos atributos generalizables. Esto último en parte ya se tuvo en cuenta al elegir cuales tomar para meter en el *.arff*.

Como se dice en el enunciado, los atributos correspondientes a información futura no se pueden utilizar como atributos de entrada para clasificar, ya que forman parte del futuro y nunca estarán disponibles a tiempo para decidir qué acción hay que realizar en el instante actual. Debido a esto, *nextScore* fue eliminado del conjunto de datos.

A continuación, con la herramienta *Select attributes* de Weka hemos hecho un pequeño estudio para ver cuáles eran los atributos más y menos relevantes. Para conocer los que más información aportaban, utilizamos el método *BestFirst* y nos dimos cuenta de que *score y pacman_dir* iban a ser importantes, lo que es lógico, pues con cada movimiento el pacman quiere mejorar al máximo posible la puntuación y para elegir ese movimiento se tiene en cuenta la dirección que lleva. Tras esto, para tener una vista más global del peso de todos ellos, con el método *Ranker* y el evaluador *infoGainAttributeEval* ordenamos los atributos del que más al que

menos aportaba con respecto a la clase. Aquí ratificamos nuestra información en cuanto a los mejores atributos y pudimos observar que *livingPacman* no era un atributo necesario, por lo que tras realizar un par de pruebas más en las que obteníamos el mismo resultado procedimos a eliminarlo. Además tiene sentido, pues ese atributo es siempre *false,* ya que el pacman no puede morir en el juego simplificado sobre el que trabajamos, y esto no aporta nada de información al clasificador. Por otro lado, tras clasificar los atributos, vemos que las direcciones de los fantasmas y la distancia hasta el punto de comida tienen poco peso para la clasificación, pero sin embargo algo aportan. Debido a esto realizamos un estudio más profundo sobre ellos, para el que utilizamos el evaluador *classifierAttributeEval* el cual nos permite seleccionar diferentes clasificadores y así poder ordenar los atributos teniendo en cuenta el clasificador que vamos a usar. Tras probar varias veces vimos que, dependiendo del clasificador que elijamos, los atributos de los que dudábamos se comportaban de mejor o peor manera, y debido a esto, nuestra decisión final fue mantenerlos en el conjunto de datos.

Por último, se probó a aplicar el filtro *removeDuplicates* el cual elimina las instancias repetidas. Sin embargo, este no causó un gran efecto en la clasificación posterior, por lo que finalmente no lo utilizamos. Utilizamos también el filtro *removeFrequentValues*, que como su nombre indica, elimina las instancias más frecuentes sobre la clase. Sin embargo, esto causó un gran efecto eliminando muchísimas instancias y dejándonos con menos de 300. Consideramos que este valor no era suficiente para entrenar un modelo y por eso tampoco lo utilizamos.

EXPERIMENTACIÓN Y EVALUACIÓN

A continuación se analizarán y compararan distintos algoritmos de clasificación. En este apartado se ha procedido de la siguiente manera:

1. Selección de algoritmos clasificadores.
2. Entrenamiento con diferentes opciones de test (*training set*, *cross-validation*, *percentageSplit*).
3. Evaluación con los dos conjuntos de test (*same/otherMaps*).

Para la selección de los clasificadores que emplearemos hemos tenido en cuenta los vistos en tutoriales previos, los vistos en clase y los que mejor resultado obtienen, tras testearlos con el *experimenter.* Teniendo en mente todo esto, seleccionamos los siguientes clasificadores: ID3, J48, RandomForest, NaiveBayes y PART.

El algoritmo ID3 se basa en árboles de decisión para clasificar la clase, al igual que el J48. El algoritmo PART emplea primero divide y vencerás, y luego en cada hoja utiliza árboles de decisión. Por otro lado, NaiveBayes, basado en el Teorema de Bayes, supone que las clases son independientes entre sí para clasificarlas correctamente. Finalmente, el algoritmo RandomForest, que es el que no hemos utilizado previamente en tutoriales anteriores, consiste en una combinación de árboles de decisión a través de un vector aleatorio.

A continuación, se ha procedido a entrenar con cada uno de los algoritmos de clasificación elegidos con las opciones mencionadas en el punto dos. Tras realizar varias pruebas vimos que

ID3 obtenía aparentemente los mejores resultados, pero esto no se repetiría a la hora de probar el modelo con un conjunto de test, debido a que este algoritmo deja instancias sin clasificar, lo que hace que los resultados empeoren a la hora de la verdad. A su vez observamos que RandomForest y J48 obtenían resultados aceptables en todas las opciones de test probadas y funcionaban mejor que el resto con la mayoría de filtros aplicados. Esto lo ratificaríamos más adelante en la sección de evaluación.

Una vez realizadas todas las pruebas de entrenamiento procedimos a realizar diferentes testeos con todos los ficheros que obtuvimos en la fase 1, es decir, tanto con los del pacman controlado por teclado como con los del pacman automático, ambos probándolos con los conjuntos de test de mapas iguales y mapas diferentes.

Para este análisis hemos incluido tablas, en las que almacenamos los resultados obtenidos y que sirven para compararlos. Estas entrelazan los algoritmos de clasificación utilizados con los filtros aplicados al conjunto de datos con mejores resultados o simplemente los empleados por necesidad o por haberlos visto en los tutoriales. Los filtros elegidos finalmente para la recopilación de datos fueron *Discretize*, *Resample*, *Normalize* y mezclas entre ellos, las cuales nos servirían para entender un poco mejor el funcionamiento de cada uno.

A continuación, se muestran las diferentes tablas con los resultados obtenidos (número de instancias bien clasificadas), el mejor de ellos está marcado en rojo y debajo de cada una de las tablas se hace un análisis de lo recopilado.

PACMAN CONTROLADO POR TECLADO

Supplied test set (*sameMaps*)

Algoritmo \ Filtro	**No filter**	**Discretize**	**Resample**	**Normalize**	**Resample-Normalize**	**Discretize-Resample**
Id3	X	33.7349 %	X	X	X	32.7273 %
J48	59.0361 %	64.4578 %	45.4545 %	50.6024 %	32.7273 %	50.303 %
Random Forest	54.8193 %	60.241 %	53.9394 %	51.2048 %	37.5758 %	64.2424 %
Naive Bayes	46.3855 %	54.2169 %	45.4545 %	50.6024 %	45.4545 %	49.697 %
Part	51.2048 %	59.6386 %	53.9394 %	48.1928 %	33.3333 %	49.0909 %

Tras probar el modelo con el conjunto de instancias de test del pacman controlado por teclado en los mismos mapas, nos reafirmamos en que J48 y RandomForest serán nuestros aliados. El primero consigue el mejor porcentaje de clasificación aplicando el filtro *Discretize*, el cual convierte un rango de atributos numéricos en norminales. El número de *bins* fue ajustado a 5

como se recomendó en los tutoriales, es decir, 5 será el número de posiciones decimales para los puntos de corte cuando se generen las etiquetas de contenedor. La opción *useBinNumbers* a *True de* tal forma que las etiquetas del conjunto de entrenamiento y las del de test coincidieran.

Por otro lado, RandomForest se comporta aceptablemente con una mezcla entre la discretización y la aplicación del filtro supervisado *Resample*. Este produce una submuestra aleatoria de un conjunto de datos utilizando muestreo. A la hora de su aplicación se ajustó el campo *biasToUniformClass* a 1. Con esto conseguimos un balanceo equilibrado de nuestro atributo clase.

Normalizar datos es una técnica que se aplica a un conjunto de datos para reducir su redundancia. El objetivo principal de esta técnica es asociar formas similares a los mismos datos en una única forma de datos. Por ello, podemos observar que los atributos numéricos se han acotado entre 0 y 1. A priori creíamos que mejoraría los resultados, sin embargo, como se puede observar, no se consigue ninguna mejora aparente.

Supplied test set (*otherMaps*)

Algoritmo \ Filtro	**No filter**	**Discretize**	**Resample**	**Normalize**	**Resample-Normalize**	**Discretize-Resample**
Id3	X	23.9264 %	X	X	X	46.875 %
J48	49.6933 %	68.0982 %	37.5 %	50.3067 %	30.625 %	50 %
Random Forest	53.3742 %	60.7362 %	52.5 %	53.3742 %	44.375 %	58.125 %
Naive Bayes	36.8098 %	33.1288 %	35.625 %	34.3558 %	31.875 %	26.875 %
Part	39.8773 %	60.1227 %	33.125 %	45.3988 %	41.875 %	45 %

Para el conjunto de test de mapas diferentes a los de entrenamiento, conseguimos resultados proporcionales a los anteriores, aunque generalmente peores porque los datos han sido recogidos en mapas dispares. Sin embargo, J48 obtiene un porcentaje de acierto en la clasificación aceptable, lo cual nos demuestra que este algoritmo es capaz de generalizar en diferentes entornos.

Part y Naive Bayes no consiguen clasificaciones muy precisas. Pero el primero, un poco mejor, será tenido en cuenta más adelante a la hora de probar el modelo pues los resultados que obtenemos en Weka son orientativos, o al menos eso nos ha demostrado nuestra experiencia.

PACMAN AUTOMÁTICO

A continuación se muestran de la misma manera los resultados obtenidos con el pacman automático. Tras realizar los mismos test, con los mismos filtros, nos damos cuenta de que los resultados son proporcionales aunque mejores a los obtenidos con el agente manual. Esto es debido a que a la hora de recoger los datos, nosotros, como humanos, hemos realizado peores decisiones que la función programada. Incluso ante una misma situación hemos podido tomar un camino diferente.

Supplied test set (*sameMaps*)

Algoritmo \ Filtro	No filter	Discretize	Resample	Normalize	Resample-normalize	Discretize-Resample
Id3	X	54.9708 %	X	X	X	55.9524 %
J48	74.269 %	68.4211 %	64.2857 %	74.269 %	61.3095 %	65.4762 %
Random Forest	80.7018 %	71.9298 %	72.0238 %	76.0234 %	65.4762 %	66.6667 %
Naive Bayes	57.8947 %	50.8772 %	58.9286 %	63.1579 %	64.2857 %	54.1667 %
Part	76.0234 %	66.0819 %	70.2381 %	65.4971 %	64.2857 %	54.7619 %

Supplied test set (*otherMaps*)

Algoritmo \ Filtro	No filter	Discretize	Resample	Normalize	Resample-normalize	Discretize-Resample
Id3	X	33.5294 %	X	X	X	47.619 %
J48	65.2941 %	73.5294 %	64.2857 %	72.619 %	60.7143 %	73.8095 %
Random Forest	61.1765 %	68.8235 %	63.0952 %	77.381 %	70.2381 %	64.2857 %
Naive Bayes	47.0588 %	54.7059 %	60.119 %	20.2381 %	61.9048 %	59.5238 %
Part	57.6471 %	68.8235 %	63.6905 %	50 %	60.7143 %	41.6667 %

Tras analizar todos los resultados obtenidos y probando algún filtro más los cuales no aparecen en la tabla por no mejorar los resultados, como el *ClassBalancer* y el *Randomize*, apreciamos que aparentemente RandomForest era el mejor clasificador, pero que sin embargo, este no era capaz de mantener sus resultados, generalmente, en las pruebas con mapas diferentes. En su lugar J48 si lo hacía, consiguiendo generalizar para diferentes escenarios.

Tras ver estos resultados, procedimos a intentar mejorarlos aún más. Para ello aplicamos el filtro *removeMissClasified,* el cual elimina las instancias que se clasifican incorrectamente. A la hora de aplicar el filtro seleccionamos el clasificador con el que íbamos a trabajar, haciendo pruebas con los que mejor nos habían funcionado, J48 y RandomForest. Para el primero los resultados mejoraron considerablemente y se eliminaron correctamente las instancias, las cuales son las que tanto nosotros como nuestra funcionalidad *chooseAction* elegía un movimiento poco óptimo. Sin embargo, para RandomForest este algoritmo no tenía ningún efecto, no eliminaba ninguna instancia. Por lo que los resultados seguían siendo los mismos que los obtenidos previamente.

Tras esto J48 gracias a sus resultados y correcta generalización se convertía en el algoritmo más adecuado, a priori, para crear nuestro modelo, a falta de su testeo en el juego.

4.- FASE 3: PREDICCIÓN

En la tercera fase se construirá un modelo de predicción sobre la puntuación que tendrá el agente en el turno siguiente. Igual que en la fase anterior, se incluirá una justificación de los algoritmos y atributos seleccionados, el tratamiento de datos llevado a cabo y un análisis de los resultados producidos por cada algoritmo, así como la justificación de la elección final.

TRANSFORMACIÓN DE DATOS

Primero se analizarán los datos y los atributos para asegurar una correcta predicción en el agente automático.

Utilizando de nuevo la herramienta *Select attributes* de Weka, hemos realizado un estudio similar al de la fase 2. En primer lugar, utilizamos de nuevo el método *BestFirst* sobre la clase nextScore, que nos informó de nuevo que los atributos *score*, *pacman_dir* y también *pacman_action* iban a ser los más importantes. A continuación, con el método *Ranker,* en este caso no podíamos utilizar el evaluador *infoGainAttributeEval*, ya que solo sirve para atributos nominales. Por ello, elegimos el *CorrelationAttributeEval* para medir la correlación con la clase y obtuvimos de nuevo que el atributo *livingPacman* no era necesario, por lo que lo eliminamos. Finalmente, para completar el estudio, usamos de nuevo el evaluador *classifierAttributeEval* que habíamos usado en la fase dos. Tras probar con distintos algoritmos, en todos ellos el atributo más importante sin lugar a dudas era el *score*, lo cual tiene sentido, ya que la puntuación del tick siguiente siempre será la puntuación actual más o menos un valor que dependerá de si se ha comido a un fantasma (o bolita de comida) o no. El resto de atributos varía en función del algoritmo escogido, así que decidimos no eliminar ninguno más.

Por otro lado, queremos destacar en este apartado que los datos obtenidos quizá no son los óptimos a la hora de realizar el entrenamiento. Nos hemos dado cuenta de que en un mapa concreto (*trappedClassic*), los fantasmas no se encierran una vez te los has comido. Además, al tratarse de un mapa muy pequeño, lo que sucede es que el pacman se puede comer a los fantasmas más de una vez, por lo que la puntuación puede llegar a ser altísima. Es por eso que

tal vez el modelo generado no sea tan bueno como podría haber sido, ya que el valor de la puntuación no termina de ser real.

En cuanto a los filtros utilizados para eliminar instancias innecesarias, de nuevo el *removeDuplicates* no fue utilizado por su mínimo impacto. Por otro lado, destacar que en este caso no es posible utilizar el filtro *Discretize*, ya que convierte todos los atributos a nominales, y al tratarse en este caso de una clase numérica (*nextScore*), no podríamos utilizar los algoritmos necesarios.

EXPERIMENTACIÓN Y EVALUACIÓN

En cuanto al análisis y la comparación de algoritmos, hemos procedido de la misma manera que en la fase anterior:

1. Selección de algoritmos de regresión.
2. Entrenamiento con diferentes opciones de test (*training set*, *cross-validation*, *percentageSplit*).
3. Evaluación con los dos conjuntos de test (*same/otherMaps*).

En primer lugar, en cuanto a los algoritmos de regresión hemos seleccionado *LinearRegression* o regresión lineal, *MultilayerPerceptron* o perceptrón multicapa, y M5P. Se trata de los algoritmos vistos en clase para tareas de regresión.

El algoritmo de regresión lineal se trata de una aproximación para modelar la relación entre una variable dependiente, en nuestro caso *nextScore*, y el resto de variables explicativas (nuestros atributos como *score*, *pacman_dir*, etc.). Por otro lado, el perceptrón multicapa se trata de una red de neuronas con distintas capas en las que se propaga la información y se van modificando los pesos para predecir el valor de la clase. Finalmente, el algoritmo M5P genera árboles de decisión similares a los producidos por ID3.

En cuanto al entrenamiento con las opciones de test mencionadas en el punto dos, queremos destacar que el valor principal que hemos utilizado para comparar los distintos algoritmos ha sido el *Root relative squared error* o error relativo. Este nos permite comparar entre modelos aunque las unidades en las que se ha medido el error en dichos modelos no coincidan. Sin embargo, el error absoluto es más útil para comparar distintas versiones del mismo modelo. Por ello, podemos concluir que, de mejor a peor, los algoritmos estarían ordenador primero M5P, luego regresión lineal y por último perceptrón multicapa. Además, nos ha llamado la atención que el perceptrón multicapa ha tardado mucho más tiempo en generarse que los otros dos, probablemente debido a la propagación hacia atrás de las distintas capas.

Para terminar, procedimos a realizar los distintos algoritmos con los ficheros de test tanto de mapas iguales como diferentes que creamos en la fase 1. Como en la fase anterior, hemos utilizado de nuevo tablas para almacenar y comparar los resultados, relacionándolos también con los distintos filtros elegidos. En este caso, tan solo hemos utilizado los filtros *Resample*, N*ormalize* y la mezcla de ambos, ya que según habíamos mencionado anteriormente, el filtro *Discretize* no es posible aplicarlo.

A continuación se muestran las diferentes tablas con los resultados obtenidos (coeficiente de correlación, error relativo y error absoluto en ese orden), el mejor de ellos está marcado en rojo y debajo de cada una de las tablas se hace un análisis de lo recopilado.

PACMAN CONTROLADO POR TECLADO

Supplied test set (*sameMaps*)

Algoritmo \ Filtro	No filter	Resample	Normalize	Resample-normalize
LinearRegression	0.9512 37.0352 % 44.6127 %	0.7656 41.2223 % 67.5591 %	0.8572 54.4016 % 55.8017 %	0.7239 87.0159 % 95.6643 %
MultilayerPerceptron	0.4296 71.9673 % 90.7128 %	0.6849 53.0644 % 74.8179 %	0.4636 101.504 % 114.0296 %	0.6271 89.1631 % 101.6717 %
M5P	0.8918 40.2132 % 58.2009 %	0.7932 34.7896 % 65.606 %	0.9115 30.4272 % 36.5551 %	0.7884 38.1494 % 46.6251 %

En esta primera tabla se representan los resultados obtenidos con las instancias de test de los mismos mapas que se utilizaron en el entrenamiento. Como se puede observar, el mejor algoritmo en este caso es el M5P aplicando el filtro *Normalize*. Esto se debe a que los datos no son tan dispares como lo serían sin filtro.

Supplied test set (*otherMaps*)

Algoritmo \ Filtro	No filter	Resample	Normalize	Resample-normalize
LinearRegression	0.9598 37.9812 % 45.8729 %	0.7439 52.7164 % 70.1374 %	0.7932 67.3423 % 66.9162 %	0.6627 102.5659 % 102.8523 %
MultilayerPerceptron	0.6227 62.2493 % 78.6234 %	0.73 55.1195 % 71.5157 %	0.6388 103.5952 % 108.1702 %	0.6184 105.859 % 108.6695 %
M5P	0.8882 44.1107 %	0.8599 46.0333 %	0.87 42.526 %	0.7423 52.4922 %

	59.7765 %	62.9536 %	48.2203 %	64.6333 %

Sin embargo, si utilizamos mapas distintos en el test y en el entrenamiento, el mejor algoritmo resulta ser el de regresión lineal. Por ello, podemos decir que este es el que más generaliza de los tres, ya que además tampoco obtiene muy malos resultados utilizando los mismos mapas.

PACMAN AUTOMÁTICO

Supplied test set (*sameMaps*)

Algoritmo \ Filtro	**No filter**	**Resample**	**Normalize**	**Resample-normalize**
LinearRegression	0.9666 19.6865 % 24.4419 %	0.9549 26.9195 % 29.6242 %	0.969 25.4453 % 27.738 %	0.9598 35.863 % 38.0324 %
MultilayerPerceptron	0.9071 36.0307 % 47.4293 %	0.9473 28.3059 % 32.0161 %	0.9119 35.474 % 44.0644 %	0.9466 36.1196 % 40.0083 %
M5P	0.9729 15.368 % 22.0672 %	0.9656 20.2551 % 26.066 %	0.9177 32.6126 % 44.8066 %	0.9289 35.7296 % 40.7397 %

En este caso, de nuevo vemos que el mejor algoritmo para los mismos mapas de test que de entrenamiento es el M5P, en este caso sin filtro. Esto se debe a que, al tratarse ahora del agente automático, los datos no se encuentran dispersos, por lo que el filtro *Normalize* no mejora los resultados. Si seleccionamos la opción *Visualize classifier errors* podemos comparar en una gráfica la disparidad y los valores atípicos de cada algoritmo.

Supplied test set (*otherMaps*)

Algoritmo \ Filtro	**No filter**	**Resample**	**Normalize**	**Resample-normalize**
LinearRegression	0.9893 21.1319 % 20.0787 %	0.9809 35.5345 % 38.6146 %	0.9708 31.6761 % 29.9965 %	0.9199 53.2691 % 52.4932 %
MultilayerPerceptron	0.9496 32.9739 %	0.8482 50.9146 %	0.7981 67.2459 %	0.7556 60.6728 %

	29.3892 %	57.1604 %	68.4616 %	64.0047 %
M5P	0.9588 33.8643 % 38.3718 %	0.9611 27.119 % 31.2839 %	0.9517 29.2697 % 29.796 %	0.8669 43.901 % 46.2461 %

También podemos ver que al utilizar distintos mapas en el fichero de test y en el entrenamiento, el mejor algoritmo es de nuevo el de regresión lineal, por lo que nos reafirmamos en que este es el que mejor generaliza en distintos escenarios.

Como conclusión, los dos algoritmos adecuados, a priori, para implementar en el juego serán el de regresión lineal, que es el que generaliza mejor en distintos mapas, y el M5P, que obtiene unos resultados notables cuando se trata de los mismos mapas y a la hora de generalizar a distintos escenarios, aun no llegando al resultado de la regresión lineal, realiza una predicción aceptable.

5.- FASE 4: CONSTRUCCIÓN DE UN AGENTE AUTOMÁTICO

En esta cuarta y última fase se explicará qué modelo se ha seleccionado y por qué para construir el agente automático en *python*, así como una breve descripción del funcionamiento del mismo. Se añadirá también una comparación con el agente manual y el agente automático realizado en el tutorial 1.

En primer lugar, de cara a la implementación del modelo, hemos seguido los pasos especificados en el enunciado. Una vez teníamos todos los paquetes instalados, el siguiente paso era importar el fichero *wekaI.py* que se suministra y que permite utilizar en *python* los modelos generados por Weka. A continuación, se arrancaba la máquina virtual de java en el fichero *bustersAgents.py*, y finalmente se invocaba al método *predict* cada vez que se quiera obtener la clasificación de uno de los modelos anteriores, es decir, en el *chooseAction*. Para esto, lo primero es generar el modelo en Weka desde el *Explorer*. Una vez lo tenemos guardado, el método *predict* recibe ese modelo, un array con los valores de la instancia a clasificar y el conjunto de entrenamiento utilizado para entrenarlo. Así, devuelve la acción de salida que devuelve nuestro modelo, es decir, la acción que realizará el pacman.

Para obtener el array de los valores de la instancia, hemos realizado distintas funciones que nos permiten insertar los atributos correctamente. Por ejemplo, los atributos nominales como *living_ghost* o *ghost_direction* deben ir entre comillas, por lo que hemos utilizado la función *str* que convierte a *string*. Por otro lado, había datos como la distancia a un fantasma o a un punto de comida, que cuando son comidos pasan a tener valor *None*. Para evitar errores, hemos creado distintos métodos para cada atributo que convierten ese *None* a -1 para que siempre sea un valor numérico.

Una vez el código estaba listo para ejecutar el juego con los modelos de clasificación que según Weka serían los mejores para seleccionar la acción, nos dimos cuenta de que estos elegían algunos movimientos ilegales, lo cual finalizaba la ejecución insatisfactoriamente. Para solucionar esto modificamos el fichero *wekaI.py* (incluido en la entrega). Dentro del mismo se cambió la función *predict* de tal manera que si la acción predicha no era legal eligiera otra, teniendo en cuenta su probabilidad devuelta por el método *distribution_for_instance,* la cual funcionaba como prioridad. De esta manera siempre se realizará el movimiento con más probabilidad que sea legal.

A continuación, una vez resuelto este problema, probamos con el modelo de J48 y el de RandomForest, sin embargo no conseguían buenos resultados, aunque en Weka parecían los mejores. Pero como ya se ha mencionado anteriormente, según nuestra experiencia es una herramienta orientativa. Debido a esto probamos a ejecutarlo con el modelo de PART, el cual daba algún que otro error, pero conseguía completar más mapas. Después de esto lo probamos con fantasmas en movimiento, que con algún modelo ayudaba, pues situaba a los fantasmas en diferentes posiciones algunas de ellas más “accesibles” para los movimientos del pacman.

Finalmente, aunque no completando la mayoría de los mapas y entrando a veces en bucles, el modelo elegido es el J48, debido a que este no ve frustrada su ejecución nunca por errores internos de Weka, como el resto que a veces si los tiene. Sin embargo, a pesar de haber obtenido el mejor porcentaje y de generalizar correctamente, este algoritmo no ha conseguido completar ningún mapa con los fantasmas estáticos.

Creemos que esto se debe a que el modelo realmente no está bien entrenado. Por otro lado, también hemos observado que el error relativo de este algoritmo es bastante alto, lo que quizá también puede influir en su desempeño en el juego.

Como se ha podido observar en las partes de análisis de los resultados, nuestro agente del tutorial 1 conseguía completar los mapas con movimientos más acertados. Esto se debe a que estaba programado con tal fin, y pese a no ser bueno del todo, seleccionamos mapas en los que funcionaba bien. Aun así, es cierto que a veces realizaba algún movimiento sin sentido. Por otro lado, el pacman manual, al ser manejado por nosotros no realizaba las acciones óptimas, pues en concreto en los mapas con fantasmas móviles es muy difícil hacerlo. Debido a esto los resultados obtenidos no han sido los mejores en cuanto al modelo, pues las instancias no eran al 100% buenas y el modelo no ha sido entrenado de la mejor manera posible. Esto quizá se pueda mejorar basando el aprendizaje del agente en refuerzo.

6.- PREGUNTAS

1.- ¿Qué diferencias hay a la hora de aprender esos modelos con instancias provenientes de un agente controlado por un humano y uno automático?

Si se aprende con instancias provenientes de un agente manual, resultará en un modelo peor que si aprendiera con un agente automático si este está medianamente bien hecho. Esto se debe a que el automático sabe en todo momento cuál será la mejor acción que puede tomar, mientras que los seres humanos cometemos más errores. Con más razón sucede en este caso, donde el pacman se mueve bastante rápido y si no estás atento mientras los controlas podrías crear instancias innecesarias, al igual que cuando se activa la opción de los fantasmas móviles que complica aún más la tarea. Por ello, la diferencia principal es que probablemente necesites realizar un tratado de datos más exhaustivo si utilizas un agente manual en vez de uno automático para garantizar un modelo correcto.

2.- Si quisieras transformar la tarea de regresión en clasificación ¿Qué tendrías que hacer? ¿Cuál crees que podría ser la aplicación práctica de predecir la puntuación?

Para transformar una tarea de regresión en clasificación, tan solo es necesario discretizar las instancias. De esta manera, se pasa de tener una clase numérica a una clase nominal. Al fin y al cabo, la regresión es la predicción de una clase numérica.

Creemos que la aplicación práctica que tiene el predecir la puntuación podría ser el averiguar siempre cuál sería la mejor acción a escoger. Si predices la puntuación siguiente, con la puntuación actual podrías saber la diferencia de puntos, y si esta se corresponde con el valor de comerte un fantasma o un punto de comida, si lo juntas con la mínima distancia, podrías averiguar la acción a realizar. Podría darse un caso en el que el pacman pudiera elegir dos caminos que a priori parecen iguales, pero uno de ellos tuviera un punto de comida y el otro no. En ese caso, si usaras solo la distancia, podrías no elegir el camino correcto y obtendrías entonces una menor puntuación.

3.- ¿Qué ventajas puede aportar predecir la puntuación respecto a la clasificación de la acción? Justifica tu respuesta.

Predecir la puntuación del pacman en el siguiente turno puede aportar diferentes ventajas las cuales no conseguimos con la clasificación de la acción.

Dado que el principal objetivo del juego con el que estamos realizando las prácticas es aumentar la puntuación al máximo posible antes de finalizar la partida, lo más correcto sería avanzar hacia la posición que más puntos nos dé, ya sea porque ahí se encuentra un fantasma o un punto de comida. Sin embargo, la clasificación de la acción, aunque generalmente consigue aumentar la puntuación, se centra en dirigirse continuamente hacia el fantasma más cercano. Esto significa

que quizá ante un camino igual de largo entre la posición del pacman y el fantasma elija uno por el que obtienes menos puntos, por ejemplo uno en el que no hay puntos de comida. Con él la predicción de la puntuación no pasaría, pues el pacman evalúa sus casillas contiguas en busca del mejor movimiento posible.

Por otro lado, el evaluar el turno en concreto y no dirigirse ciegamente a una posición dentro del mapa cuenta con la ventaja de aprovechar al máximo cada decisión. Puesto que aunque a veces se realice un camino más largo, debido a la cantidad de puntos que se obtienen al comerse fantasmas y comida respecto a los puntos que se restan por tick, finalmente al acabar la partida se contará con una puntuación más elevada. Además, permitirá en gran medida no entrar en posibles "callejones sin salida" de paredes dentro del mapa.

4.- ¿Crees que se podría conseguir alguna mejora en la clasificación incorporando un atributo que indicase si la puntuación en el instante actual ha ascendido o ha bajado?

Creemos que no, pues aunque tenga un objetivo similar al que se consigue con la predicción de la puntuación, en muchos mapas se disminuye la puntuación en una unidad en una gran cantidad de turnos, debido a que hay mucha distancia entre fantasmas o puntos de comida y pacman. Sin embargo, esto no significa que se haya elegido la acción incorrecta o menos eficiente, simplemente que hasta llegar a una fuente de puntos hay varios movimientos que no mejoran la puntuación. De hecho, quizá esto también se podría comprobar sabiendo la puntuación en cada instancia, por lo que no aportaría nada nuevo.

7.- CONCLUSIONES

Una buena obtención y transformación de datos y su correcto tratamiento pueden ayudar mucho a la hora del aprendizaje automático y gracias a este filtrado se conseguirán modelos mucho más precisos. A su vez, si las instancias han sido extraídas de un proceso manual, estas pueden contener una información menos útil que si se tratase de los resultados obtenidos de un agente automático, programado específicamente para tomar la decisión más correcta.

El modelo obtenido sirve para generar un proceso automático basado en la experiencia, el cual si ha sido correctamente entrenado conseguirá una precisión muy notable a la hora de tomar decisiones en la tarea asignada. Tras realizar la práctica, nos hemos dado cuenta de la gran utilidad que tiene el aprendizaje automático, además de en los juegos, en otros dominios como el de los coches autónomos. Estos, entre otras cosas, su funcionalidad es elegir la acción que van a realizar en base a distintas observaciones del entorno físico en el que se desplazan, como puede ser la distancia a otros objetos.

El principal problema encontrado en esta práctica ha sido el de saber cuándo parar de experimentar, pues entendemos que si se tratase de un trabajo real, esta tarea sería mucho más extensa y exhaustiva. Sin embargo, al tratarse de una práctica universitaria suponemos que el principal objetivo no es encontrar el mejor modelo posible, sino simplemente entender los

realizados. Por otro lado, en la fase final del proyecto, hemos tenido varios problemas al ejecutar nuestro código, principalmente provenientes de Java y del predictor de Weka.

En definitiva, una práctica útil para conseguir entender y fijar los contenidos vistos en las clases teóricas, con una estructura progresiva que sirve para ello. En contraposición, creemos que alguna indicación más en cuanto a que es lo que se va a evaluar ayudaría mucho a los alumnos a la hora de tomar decisiones. Al igual que la inclusión de ayudas para los posibles errores que puedan darse en la fase 4 del trabajo.