Universidad Carlos III de Madrid

Aprendizaje Automático

Grado en Ingeniería Informática

Grupo 83

Práctica 2: Aprendizaje basado en instancias

Autores:
Daniel Medina García
Alejandro Rodríguez Salamanca

6 de abril de 2016

Índice

Introducción

El presente documento contiene la memoria del trabajo realizado para esta segunda práctica de Aprendizaje Automático. En ella, el equipo ha utilizado el aprendizaje basado en instancias, haciendo uso de la técnica de *clustering* para poder implementar funciones de afinidad y agilizar así la clasificación. Continuamos así nuestra introducción al aprendizaje no supervisado, en oposición al aprendizaje supervisado visto en ejercicios anteriores.

1. Recogida de información

Tomamos como base la toma de datos de la anterior práctica, para construir sobre ella ligeras mejoras que en nuestra opinión optimizaban la información almacenada. Así, los siguientes datos fueron recopilados en tiempo de ejecución:

- `score` : atributo numérico que contiene la puntuación instantánea. Su rango tiene cota una superior marcada por el número de fantasmas con el que se jueguen (cada fantasma aporta 250 puntos) y no tiene cota inferior, ya que cada turno que pase *PacMan* sin comer perderá un punto.
- `ghosts-living` : atributo numérico que lleva la cuenta de los fantasmas que quedan vivos en el momento actual. Su valor oscila entre 1 (sólo queda un fantasma por cazar) y 4 (situación inicial).
- `distance-ghost`$_i$: cuatro atributos numéricos que hacen de vector de distancias a los respectivos fantasmas en este turno.
- `prev-distance-ghost`$_i$: cuatro atributos numéricos que expanden la información desde el turno anterior para evaluar el acercamiento o distanciamiento a los fantasmas. Todas las distancias tienen como mínimo 1 y la diagonal del tablero como máximo.
- `pos<axis>` : coordenadas de PacMan, en forma de número entero. Sus valores se encuentran dentro del rango de los ejes X e Y del mapa de juego.
- `direction` : atributo enumerado que indica el anterior movimiento de PacMan, comprendido entre los cinco posibles valores *North, South, East, West* y *Stop*.
- `wall-<direction>` : 4 atributos enumerados en forma booleano que indican si hay o no un muro en las posiciones contiguas a PacMan.
- `move` : Acción que toma el agente en el turno actual, con posibles valores *North, South, East, West* y *Stop*, si bien esperamos que esta última no sea tomada nunca.
- En continuación de la parte de predicción de la *Práctica 1*, almacenamos seguidamente todos los atributos mencionados hasta ahora a tres turnos vista. Estos atributos pueden verse en el archivo como <atributo>N, indicando que es el atributo del turno futuro.
- `fx` : valor de la función de evaluación, que explicaremos más adelante.

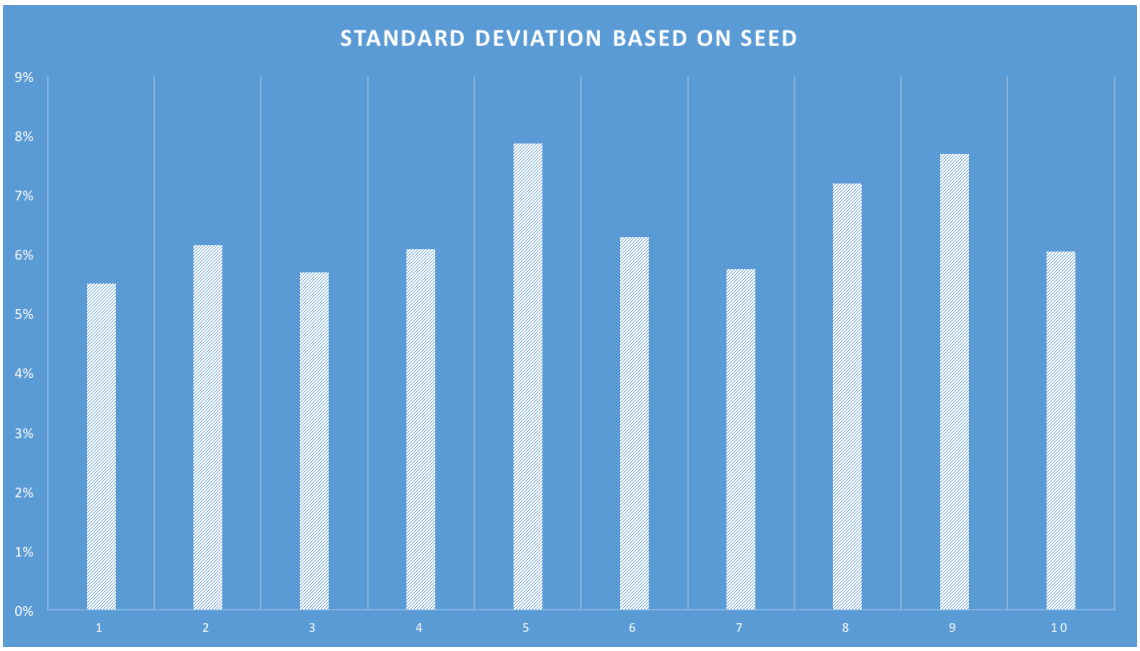
Para optimizar la recogida de datos, además:

- Si el movimiento que realiza PacMan es *Stop*, no lo escribimos. Esto es debido a que no queremos que nuestro agente se quede parado en mitad de la partida.
- Si el valor de la función es menor que en el turno anterior, no guardamos la instancia, ya que consideramos que el movimiento que se ha realizado no ha sido bueno.

2. Clustering

Tras probar todos los diferentes algoritmos de *clustering* ofrecidos por *Weka*, hicimos un primer filtro con aquellos que nos daban un número manejable de *clusters* (o se podía configurar dicho número) para evitar aquellos que generaban demasiados (menos de un 5 % de pertenencia) o insuficientes (menos de 4). Esta primera selección nos dejó con *Cobweb*, *EM*, *FarthestFirst* y *SimpleKMeans*. Comparando los algoritmos, buscamos dos propiedades: equilibrio entre los *clusters* y "estabilidad" entre ejecuciones al modificar los parámetros (i.e. semilla u otras constantes). Esta comparativa nos hizo decantarnos por *SimpleKMeans* y *EM*, pues los porcentajes de pertenencia a cada *cluster* eran más parecidos entre sí y distintas semillas resultaban en *clusters* de dimensiones similares.

Si bien los resultados eran parecidos entre estos dos algoritmos, el elevado coste en tiempo para elaborar el *clustering* con *EM* nos hizo decantarnos por *SimpleKMeans*. Mostramos a continuación la justificación de nuestra decisión, donde observamos el equilibrio conseguido con este algorimo de *clustering* y su estabilidad ante el cambio de la semilla. Cabe destacar que con los otros algoritmos encontramos variaciones muy superiores (e.g. entre 11 y 17 % con *FarthestFirst*, alcanzando el 15 % con *Cobweb*), alejadas de la media de 6 % obtenida con *SimpleKMeans*.



Para potenciar la eficacia de la clusterización, probamos a normalizar los datos. Sin embargo, los resultados obtenidos fueron los mismos. Como la normalización de los datos dificultaba la clasificación de nuevas instancias desde el wrapper de *Weka* para `Python`, decidimos excluir este y otros filtros de preproceso de los datos.

Para determinar si los movimientos tomados habían sido o no buenos, elaboramos una función cuyo incremento entre turnos supondría un rendimiento productivo. Esta función, que consta de tres sumandos con un coeficiente a modo de peso para evaluar distintos aspectos del agente, nos ayudará a descartar las instancias que no consideremos que ayuden para clusterizar correctamente:

$$f(x) = 0,5 * (\frac{distancia_{cercano}}{diagonal})^{-1} \ + \ 0,2 * (\frac{distancia_{media}}{diagonal})^{-1} \ + \ 0,3 * \frac{fantasmas_{muertos}}{fantasmas_{Iniciales}}$$

- En primer lugar, hacemos visible el acercamiento al fantasma más cercano normalizando la distancia a la que se encuentra PacMan de éste. Tomar la inversa de este dato nos asegura que un incremento en la función supone un movimiento acertado.
- Para evaluar el rendimiento a largo plazo, introducimos también la media de distancias normalizadas a los fantasmas. De nuevo, la inversa nos proporciona el dato que queremos incrementar.
- Por último, no podemos olvidarnos del objetivo principal del PacMan. Así, incluimos la cuenta de los fantasmas comidos en el último turno.

Al considerar de distinta imporancia los diferentes aspectos evaluados, incluimos los pesos de **0'5, 0'2 y 0'3** respectivamente a los sumandos previamente mencionados.

Las instancias pertenecientes a cada *cluster* son almacenadas en un array bidimensional: la primera dimensión indica el *cluster* y la segunda indica el índice de cada instancia dentro del cluster. A través del wrapper clusterizamos los datos de entrada al inicializar el agente automático, y según resulten en uno u otro *cluster* son añadidos a un u otro array.

En cuanto a la selección del *cluster* para cada instancia utilizamos, como anteriormente mencionamos, el algoritmo *SimpleKMeans* proporcionado por *Weka*, con 10 *clusters* y semilla igual a 4. El número de *clusters* se ha escogido teniendo en cuenta que tenemos cuatro movimientos posibles, y varias situaciones posibles por las cuales tomar dichos movimientos, por lo que consideramos que 10 podía ser un número suficientemente significativo como para separar los datos, dato que posteriormente hemos podido comprobar con *Weka*, obteniendo los siguientes resultados:

Cluster	Porcentaje
0	8 %
1	10 %
2	15 %
3	5 %
4	7 %
5	7 %
6	16 %
7	13 %
8	11 %
9	10 %

3. Generación del agente automático

Una vez generados los *clusters* en los cuales separamos los datos entre los que clasificaremos las instancias nuevas, pasamos a lo que realmente integra la elección de la acción a tomar.

Para generar nuestro agente, hemos usado el *wrapper* de *Weka* para `Python`.

1. Primero, cargamos el archivo `.arff` que contiene nuestras instancias, y haciendo uso de la clase `Clusterer` de `Weka`, generamos el *cluster*.

2. Después, sobre los datos previamente cargados en el programa, aplicamos el filtro `AddCluster`, que asignará a cada instancia su correspondiente *clusters*, para que luego podamos comparar estas instancias con las nuevas.

3. Tras este proceso, guardamos en una matriz las instancias separadas por *cluster*, tal y como hemos explicado previamente.

4. Finalmente, cada vez que recibimos una nueva instancia en una partida, llamamos al método `cluster_instance` de la clase `Clusterer`, al cual le pasamos la instancia, y nos devolverá el index del cluster al que ésta pertenece.

Una vez sabemos a qué *cluster* pertenece nuestra nueva instancia, recorremos la fila correspondiente de la matriz previamente descrita, comparando todas las instancias contenidas en el *cluster* con nuestra nueva instancia haciendo uso de la función de similitud que explicaremos a continuación, obteniendo un valor numérico que indica el parecido que existe entre las dos instancias.

La **función de similitud** implementada asigna pesos a los diferentes atributos guardados de cada instancia para determinar un punto para cada instancia cuya distancia euclídea ponderada con otra instancia determinará cómo de afín le es. Repartimos los pesos según lo que consideramos "áreas de conocimiento", o grupos de atributos que contienen una misma información. De esta forma, agrupamos la puntuación, las distancias a los fantasmas, la posición de *PacMan*, la dirección y la existencia de muros alrededor, quedando de la siguiente forma:

$$fSimilitud(instancia) = 0{,}2 * fantasmas_{vivos} \ + \ 0{,}2 * \sum_{i=0}^{fantasmas_{vivos}} distancia_i \ + \ 0{,}2 * (pos_X + pos_Y) \ + \ 0{,}2 * direccion \ + \ 0{,}2 * \sum_{i=0}^{dimension_{tablero}} hayMuro_i$$

- Sumando 1: número de fantasmas que están vivos.
- Sumando 2: suma de las distancias a los fantasmas vivos
- Sumando 3: suma de las coordenadas X e Y de *PacMan*.
- Sumando 4: valor numérico que asignamos a cada una de las diferentes direcciones, *Stop* (4), *East* (3), *West* (2), *South* (1), y *North* (0).
- Sumando 5: sumatorio sobre la lista de muros alrededor de *PacMan*. Si existe un muro (*True*), sumamos 1, si no existe (*False*), el valor se mantiene.

3.1. Preguntas propuestas:

¿Por qué ha sido útil realizar *clustering* previa de las instancias?

El uso de *clusters* permite ahorrar bastante tiempo en comparaciones para la clasificación. Al tener *clusters* ya hechos, sólo se compara la instancia nueva con aquellas que pertenezcan al mismo *cluster* en lugar de con todo el set de entrenamiento, sabiendo que la instancia más cercana se hallará en dicho cluster.

¿Por qué es importante usar pocos atributos en técnicas de aprendizaje no supervisado?

El aprendizaje no supervisado busca redundancias entre las instancias que indiquen algún criterio con el que diferenciar grupos que permitan agrupar estos ejemplos en grupos. Encontrar redundancias es un proceso cuya dificultad crece de forma exponencial con respecto al número de atributos que se tienen en cuenta, por lo que minimizar el número de atributos con los que se hacen los *clusters* optimizará notablemente el proceso.

¿Qué ventaja tiene el uso del aprendizaje basado en instancias con respecto al visto en la práctica 1?

El aprendizaje no supervisado tiene como ventaja que no requiere datos previamente etiquetados sino que es el propio algoritmo el que busca el *grupo de datos* al que pertenece sin necesidad de saber los grupos que existen.

¿Consideras que el agente funcionaría mejor si se introdujesen más ejemplos? ¿Por qué?

El proceso de creación de *clusters* se basa en la búsqueda de redundancia entre instancias, por lo que un conjunto de entrenamiento más grande podría ayudar a formar *clusters* más informados. Cuantas más instancias tenga el proceso de clusterización más definidos estarán los grupos si los hay; si, por el contrario, los grupos no están definidos con un numero cuantioso de instacias, puede significar que los atributos recopilados no aporten la suficiente información como para separar las instancias como queremos y debamos buscar otro enfoque distinto para poder diferenciar instancias.

4. Evaluación de los agentes

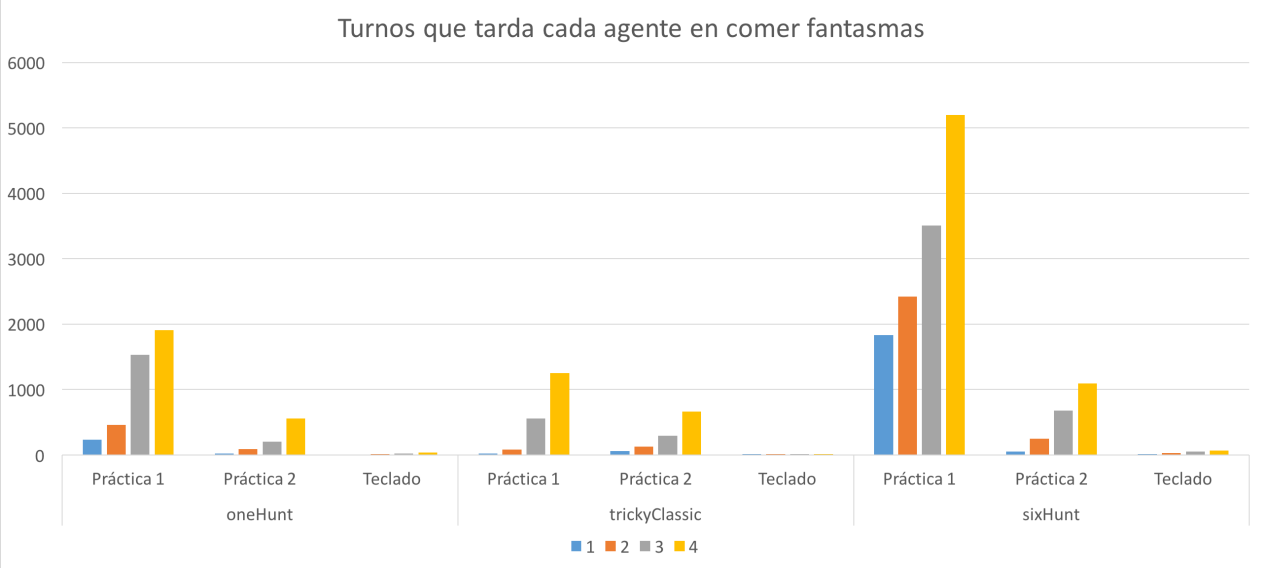
Una vez generado el nuevo agente, llegó la parte más esperada. Realmente nos intrigaba ver cómo respondería el nuevo agente en acción en comparación al desarrollado en la práctica anterior al haber utilizado técnicas tan distintas para decidir qué acción tomar.

Para evaluar el agente generado, se puso a jugar partidas a los tres agentes en distintos escenarios para poder comparar distintas situaciones y comprender globalmente los resultados.

Para ejecutar el agente, es necesario seguir lo siguientes pasos:

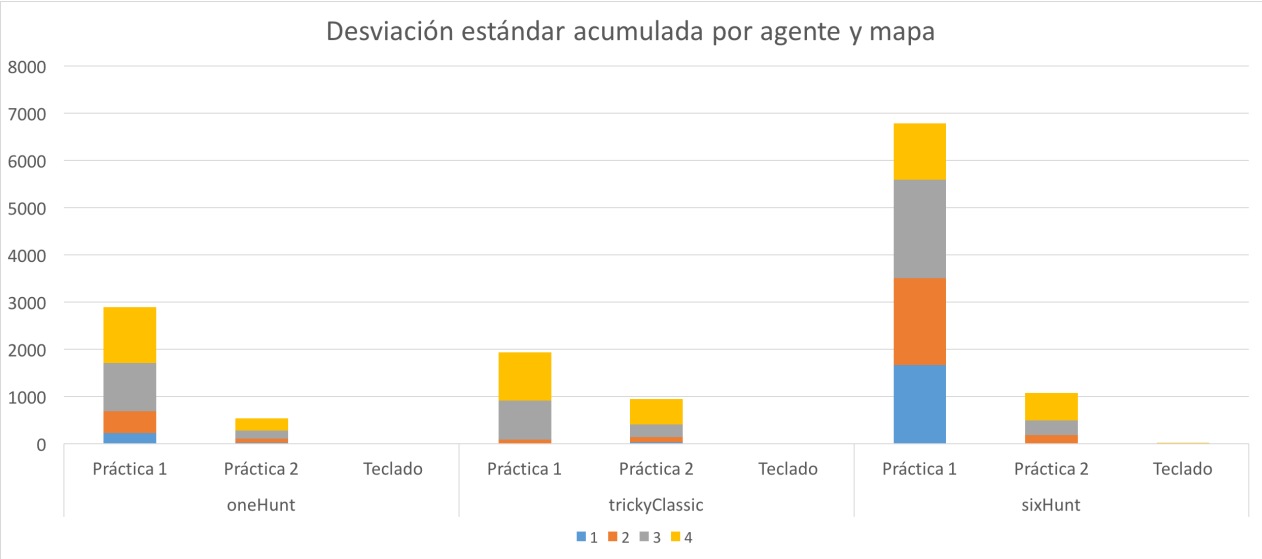
1. Instalar las dependencias y el wrapper de *Weka* para `Python`.[1]
2. Ejecutar el comando `python busters.py -p ClusteredAgent`

Así, nuestro agente buscará en la carpeta `data/` el fichero `game_toCluster.arff` para crear el *cluster* y poder clasificar las nuevas instancias. A continuación se muestra una gráfica que ilustra cuántos turnos tardó cada agente en llegar a todos los fantasmas:

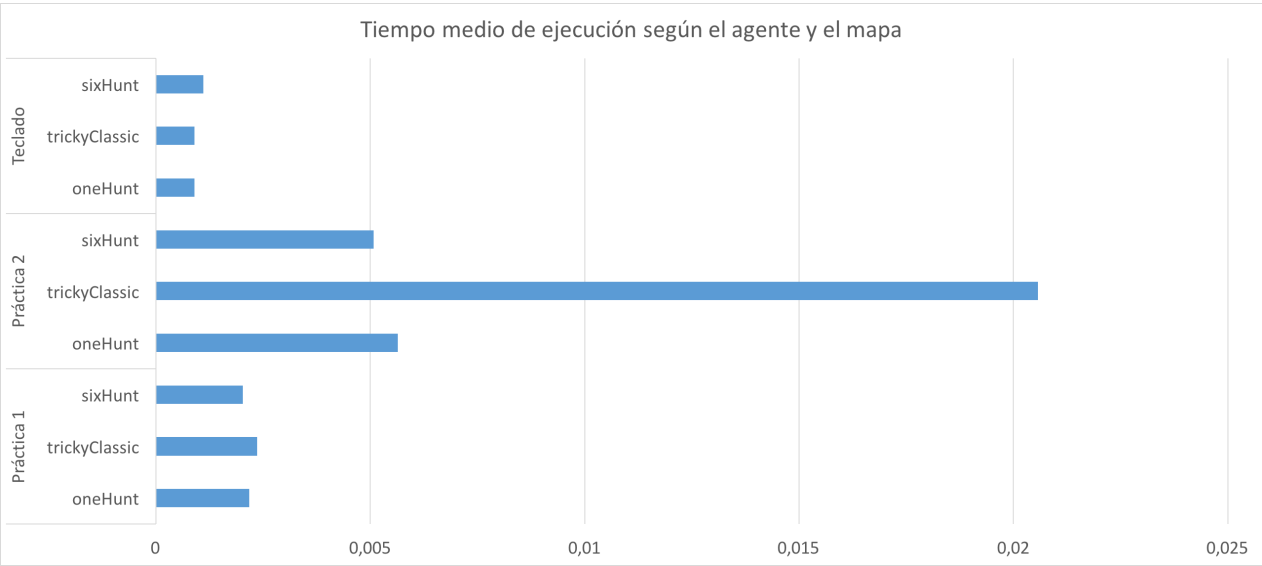


El agente de teclado (humano), al contar con la información extra de dónde está cada uno de los fantasmas en cada momento, es capaz de obtener un rendimiento bastante mejor (rozando el óptimo). Sin embargo, comparando los dos agentes automáticos observamos una **mejora muy clara** del nuevo agente respecto al anterior, de en torno al 60 %. En la siguiente gráfica mostramos la desviación estándar de los resultados de la gráfica anterior para poder ver con perspectiva los datos obtenidos. En ella, podemos ver que el agente de la *Práctica 1* se aleja considerablemente de un comportamiento determinista, con grandes diferencias entre los resultados de cada partida. Esto confirma también que nuestro nuevo agente toma decisiones mucho más informadas, consiguiendo una desviación mucho menor en sus resultados.

[1] https://pythonhosted.org/python-weka-wrapper/install.html



Sorprendidos con esta mejora tan llamativa, pasamos a comparar el tiempo empleado en cada turno para tomar la decisión que indicará a PacMan qué acción tomar. La siguiente gráfica compara el tiempo medio de respuesta de cada agente:



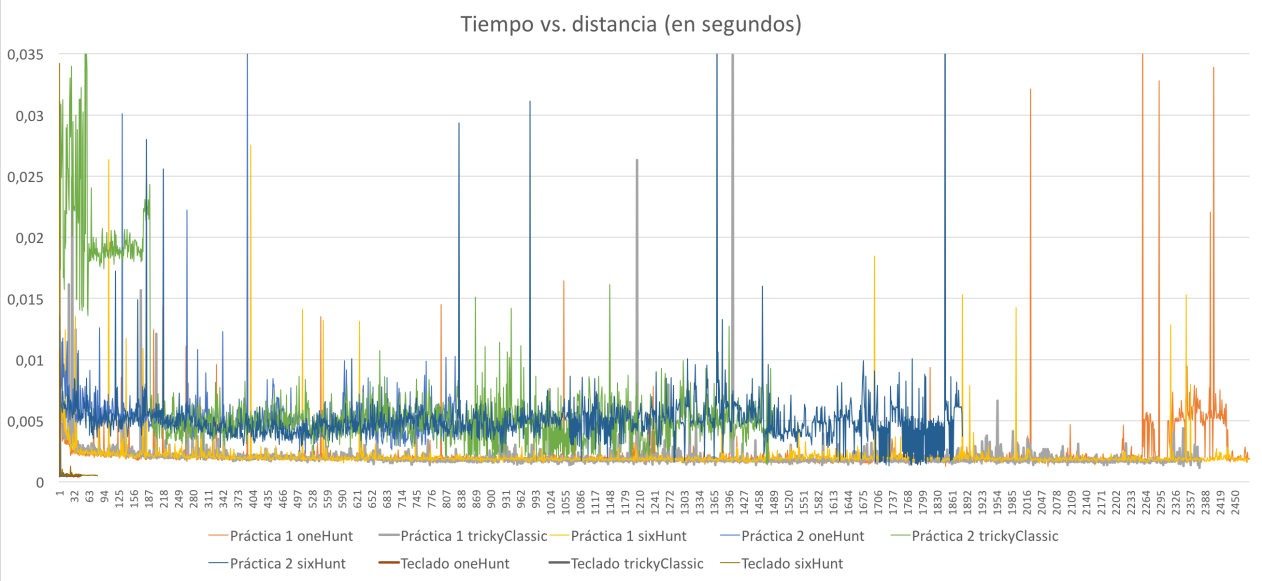
Curiosamente, si bien el rendimiento comprobábamos que se había incrementado considerablemente con el agente que utiliza aprendizaje no supervisado, ahora observamos una **eficiencia mucho menor** en esta implementación. La causa de esta contrapartida es la cantidad de cómputo realizada por este agente en comparación con los otros dos:

- El agente **humano** toma como tiempo de computación tan sólo lo que tarda el proceso de `Python` en recibir e interpretar la tecla del teclado que ha pulsado el humano, siendo este un intervalo ínfimo, despreciable.

- El agente de la **Práctica 1** genera un árbol de clasificación al iniciar el juego con los datos que tenía antes de empezar una partida y, una vez generado, lo utiliza para clasificar cada nueva instancia. De esta forma, el tiempo real de cómputo se encuentra en la inicialización del agente, dejando para cada turno tan sólo el "descenso" del árbol de la nueva instancia que "caerá" en una clase o en otra.

- Sin embargo, el agente de esta **Práctica 2** compara, en cada turno, la nueva instancia con todas aquellas contenidas en el *cluster* al que se le ha asignado para ver cuál de ellas se le parece más. Esto requiere calcular la función de similitud por cada una de las instancias del clúster, lo que supone un tiempo muy elevado en cuanto a cómputo.

Más allá de los análisis previamente mostrados, la siguiente gráfica ilustra el tiempo de ejecución versus la distancia recorrida por el agente:



En esta última comparación observamos cómo los tiempos de ejecución se mantienen bastante constantes a lo largo de toda la ejecución del programa. Mostramos sólo hasta el turno 2500 ya que sólo el agente de la *Práctica 1* continúa más allá y no es éste el que nos interesa evaluar.

5. Conclusiones

El modelo obtenido en esta práctica puede ser útil como parte del agente automático para el juego de PacMan original, así como para otros juegos donde el agente tenga que moverse y perseguir objetivos.

Al realizar la práctica hemos entendido campos donde podrían usarse técnicas de aprendizaje automático como el cálculo de rutas en tiempo real para un vehículo que se mueve por un entorno.

Uso de *Weka* en `Python`

El mayor beneficio que nos ha aportado usar el wrapper de *Weka* a la hora de generar nuestro agente, es que si en algún momento queremos cambiar el algoritmo de *clustering*, sólo tenemos que editar una línea de código. Por otra parte, si lo que queremos es obtener más datos para realizar el *cluster*, es tan sencillo como añadirlos a nuestro archivo `.arff`. Gracias a esto, hemos podido cambiar y probar diversas opciones y algoritmos sin necesidad de que la implementación de nuestro agente haya variado, lo que nos ha ahorrado mucho tiempo.

Problemas encontrados

Como de costumbre en tareas de aprendizaje automático, los problemas surgidos durante el desarrollo de la *Práctica 2* han solido estar relacionados con los "palos de ciego"necesarios para llegar a resultados concluyentes. Bajo el lema de "prueba, error y aprendizaje", el equipo trabajó para afinar los resultados tanto como fuese razonable.

Además, el wrapper de *Weka* nos ha vuelto a ocasionar algún contratiempo, tal y como en la práctica anterior, por la falta de ejemplos disponibles y su escasa documentación.

Comentarios personales

Nos ha parecido muy interesante cómo la combinación de diferentes técnicas da como resultado una búsqueda más eficiente de soluciones, como en esta práctica el procesamiento de asociación en *clusters* de los datos para reducir el dominio de búsqueda para la clasificación de una instancia. Este encuentro con el aprendizaje no supervisado nos ayudará a tener más herramientas a la hora de solucionar futuros problemas.