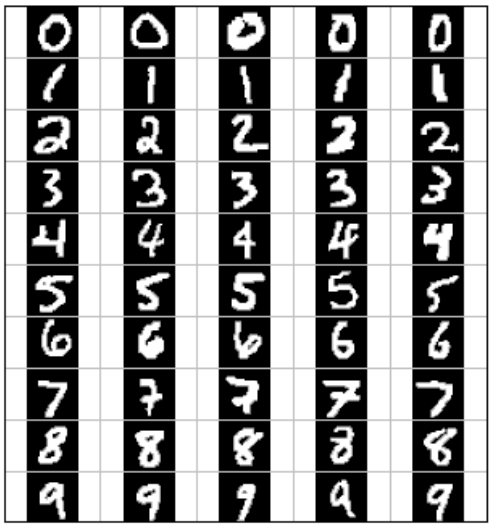
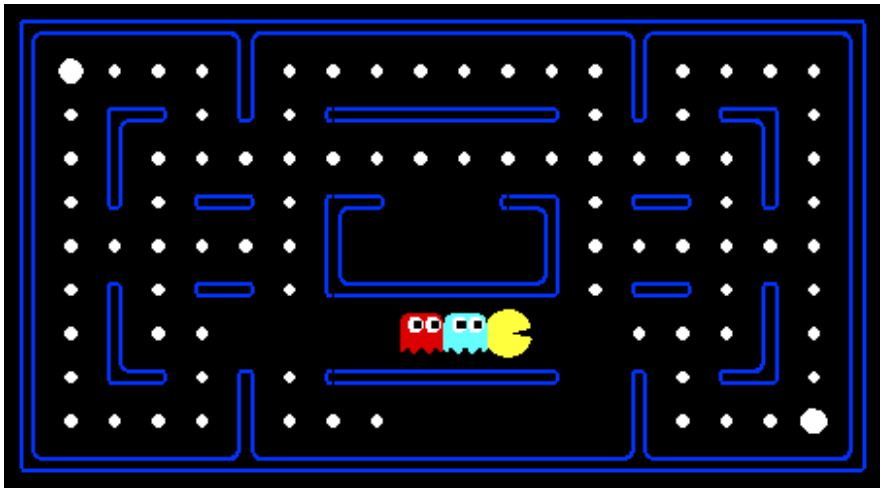
PL2: Clasificación

Tabla de contenido



Which Digit?



Which action?

Introducción

En este proyecto, diseñará tres clasificadores: un clasificador perceptrón, un clasificador MIRA y un clasificador perceptrón ligeramente modificado para la clonación de conducta. Probaréis los dos primeros clasificadores en un conjunto de imágenes de dígitos escritas a mano escaneadas, y el último en conjuntos de juegos pacman grabados de varios agentes. Incluso con funciones simples, sus clasificadores podrán desempeñarse bastante bien en estas tareas cuando reciban suficientes datos de entrenamiento.

El reconocimiento óptico de caracteres (OCR) es la tarea de extraer texto de las fuentes de imagen. El conjunto de datos en el que ejecutaréis los clasificadores es una colección de dígitos numéricos escritos a mano (0-9). Esta es una tecnología comercialmente útil, similar a la técnica utilizada por la oficina de correos de los Estados Unidos para enrutar el correo por códigos postales. Hay sistemas que pueden funcionar con una precisión de clasificación superior al 99% (consultar LeNet-5 para ver un sistema de ejemplo en acción).

La clonación es la tarea de aprender a copiar un comportamiento simplemente observando ejemplos de ese comportamiento. En este proyecto, utilizará esta idea para imitar a varios agentes pacman utilizando juegos grabados como ejemplos de entrenamiento. Su agente ejecutará el clasificador en cada acción para intentar determinar qué acción tomaría el agente observado.

Podemos descargar todo el código y los archivos de soporte con el nombre **clasificacion.zip**:
NOTA: En este proyecto usaremos Python 2.

Ficheros con datos:

data.zip	Contienen los datos sobre reconocimiento de dígitos

Ficheros que editaremos usando Python 2:

perceptron.py	Fichero donde escribirás el código de tu clasificador perceptrón.
mira.py	Fichero donde escribirás el código de tu clasificador MIRA.
answers.py	La respuesta a la pregunta 2 ira aquí.
perceptron_pacman.py	Fichero donde escribirás el código para clonar el comportamiento de un experto.

Ficheros que puede interesar mirar:

classificationMethod.py	Super clase Abstracta para los clasificadores que vas a emplear.
samples.py	I/O código para leer los datos.
util.py	Código con herramientas útiles.
mostFrequent.py	Un clasificador básico que clasifica todas las instancia con la clase más frecuente.

Evaluación:
No se pueden cambiar los nombres de las funciones o clases proporcionadas dentro del código, o causará conflictos en el autograder.

Ayuda:
¡No estas solo/a! Si os encontráis atascados en algo, debéis poneros en contacto con los profesores del curso para obtener ayuda. Las tutorías y el foro de discusión están a vuestra disposición; por favor usadlos. Se quiere que estos proyectos sean gratificantes e instructivos, no frustrantes y desmoralizadores. Pero no sabemos cuándo o cómo ayudar a menos que nos lo solicitéis.

Pregunta 1 (4 puntos): Perceptrón

Se os provee de un esquema básico que encontraréis en el perceptron.py. Aquí deberíais de rellenar la función ***train***.
Dada la lista de características ***f,*** el perceptrón computa (predice) la clase ***y'*** en base al producto escalar $\boldsymbol{f} \cdot \boldsymbol{w}$. Formalmente, dado el vector de características ***f*** el score obtenido para cada clase será:

$$score(f, y'') = \sum_i f_i \cdot w_i^{y''}$$

NOTA: ***y''*** representa una de las posibles clases o predicciones, mientras que ***y*** representa la verdadera clase y ***y'*** representa la que nuestro sistema calculará como su predicción favorita).

Después se seleccionará la clase ***y''*** que tenga el mayor valor, *score* (obtenido del producto escalar).

Ajustando los pesos

En el perceptrón multiclase básico, iremos recorriendo las instancias o ejemplos de entrenamiento, una instancia cada vez. Cuando estemos tratando la instancia (*f* , *y*), y como acabamos de explicar, seleccionaremos como nuestra predicción la clase ***y''*** con mayor *score*:

$$y' = \underset{y''}{argmax}\ score(f, y'')$$

Después, se compará ***y'*** con la clase verdadera ***y***. Si ***y' = y***, la instancia se ha clasificado correctamente, y por lo tanto no hay que llevar a cabo ninguna actualización. Por el contrario, si la predicción ***y'*** no se corresponde con ***y*** tendremos que actualizar los pesos. Eso implica que ***w·y''*** donde ***y''*** *es* ***y*** debería haber obtenido una puntuación ***f*** más alta, y por el contrario ***w·y'*** debería haber obtenido una puntuación ***f*** más baja, y prevenir así que vuelva a ocurrir el mismo error en el futuro. Así que los pesos asociados a las dos clases implicadas se actualizan de la siguiente manera:

$$\boldsymbol{w}_y = \boldsymbol{w}_y + \boldsymbol{f}$$
$$\boldsymbol{w}_{y'} = \boldsymbol{w}_{y'} - \boldsymbol{f}$$

Ejecuta tu código empleando:

python dataClassifier.py -c perceptron

Observaciones:

- El comando debería obtener una tasa de acierto entre un 40% y un 70%.
- Uno de los problemas del perceptrón es que es muy sensible a; cuantas iteraciones se realizan sobre los ejemplos de entrenamiento, el orden en el que se presentan los ejemplos de entrenamiento (lo mejor es que sea aleatorio), la normalización de las características ...
- El presente código está configurado para realizar 3 iteraciones. Podéis modificar el número de iteraciones a través de la opción -i iterations. Emplea diferentes números de iteraciones y comprueba como varían los resultados. Si esto fuese un experimento real, deberíais de emplear la tasa de acierto sobre el conjunto de desarrollo para decidir cuando para de entrenar (cuantas iteraciones), pero para este ejercicio se ha simplificado la tarea y no hay que hacer.

Pregunta 2 (1 punto): Análisis del perceptrón

Visualizando los pesos

El perceptrón y técnicas similares de clasificación, son frecuentemente criticadas porque es difícil interpretar los pesos que se aprenden. Se os pide implementar una función que identifique las características más significativas (las que mayor peso tengan) para una determinada clase.

Pregunta

Rellena ***findHighWeightFeatures(self, label)*** en perceptron.py. Debería devolver las lista de las 100 características con mayor peso para una determinada clase. Ejecutando la siguiente llamada se mostrarán por pantalla los 100 pixeles (características) con mayor peso:

python dataClassifier.py -c perceptron -w

Empléalo y responde a la siguiente pregunta. Teniendo en cuenta lo que se muestra en la pantalla, y la secuencia de imagenes que se muestran en la siguiente página, ¿cual de las siguientes secuencias de pesos representa mejor lo aprendido por el perceptrón?
Responde a esta pregunta en el método q2 de answers.py, devolviendo una 'a' o 'b'.

a)

b)

Pregunta 3 (6 puntos): MIRA

Se tiene el esqueleto para implementar MIRA en mira.py. MIRA es un clasificador que se asemeja mucho a los vectores de soporte (support vector machine, SVM) y al perceptrón. Rellena la función trainAndTune (para ello revisa las transparencias de la teoría) en mira.py. Este método empleará diferentes valores de *C* que se encuentran en Cgrid. Para cada valor de *C*, se calcularán los pesos, y al después, utilizando los datos de ***validationData*** junto con ***validationLabels*** que se nos proporcionan como parámetros, haremos la llamada a la función ***classify()***, para saber cual de los pesos de los distintos C son los mejores y quedarnos con ellos al final. En caso de empate, escoja el valor más bajo de *C*. Pruebe su implementación de MIRA con:

python dataClassifier.py -c mira --autotune

Observaciones:

- Pasar los datos self.max_iterations veces durante el entrenamiento.
- Almacenar los pesos aprendidos utilizando el mejor valor de *C* al final en self.weights, para que estos pesos se puedan usar para probar en el clasificador.
- Para usar un valor fijo de *C* = 0.001, elimine la opción --autotune del comando anterior.
- La tasa de acierto de validación cuando se usa --autotune debe estar alrededor del 60%.

Pregunta 4 (4 puntos): Clonación de comportamiento

Has construido dos tipos diferentes de clasificadores, un clasificador perceptrón y otro MIRA. Ahora implementarás una versión modificada del perceptrón para aprender de los agentes pacman. En esta pregunta, completará el método de entrenamiento en perceptron_pacman.py. Este código debe ser similar al que ha escrito en perceptron.py.

Para esta aplicación de clasificadores, los datos serán estados, y las etiquetas para un estado serán todas las acciones legales posibles desde ese estado. A diferencia del perceptrón para dígitos, todas las etiquetas comparten un solo vector de peso ***w***.

Para cada acción, calcule la puntuación de la siguiente manera:

$$score(s,a) = w \cdot f(s,a)$$

Luego, el clasificador deberá de escoger la etiqueta que reciba la puntuación más alta:

$$a' = \underset{a''}{argmax}\, score(f, a'')$$

Las actualizaciones de los pesos se producen de manera muy similar a la de los clasificadores estándar. En lugar de modificar dos vectores de peso separados en cada actualización, los pesos para las etiquetas reales y previstas, ambas actualizaciones se producen en los pesos compartidos de la siguiente manera:

$w = w + f(s, a)$ # para la acción correcta

$w = w - f(s, a')$ # para la acción predecida

Pregunta:

Rellenar el método ***train*** en perceptron_pacman.py.
Ejecutarlo llamando a:

python dataClassifier.py -c perceptron -d pacman

Este comando debería proporcionar validación y precisión de prueba **70%.**

Entrega

Para presentar el proyecto, se debe entregar:

- Ficheros **answers.py, perceptron.py, mira.py y perceptron_pacman.py**
- **Documentación** en la que se presentan los problemas abordados y su solución, con una explicación razonada de la solución empleada (estructuras de datos, aspectos reseñables, …). No hay que explicar el código, si no, el por qué!