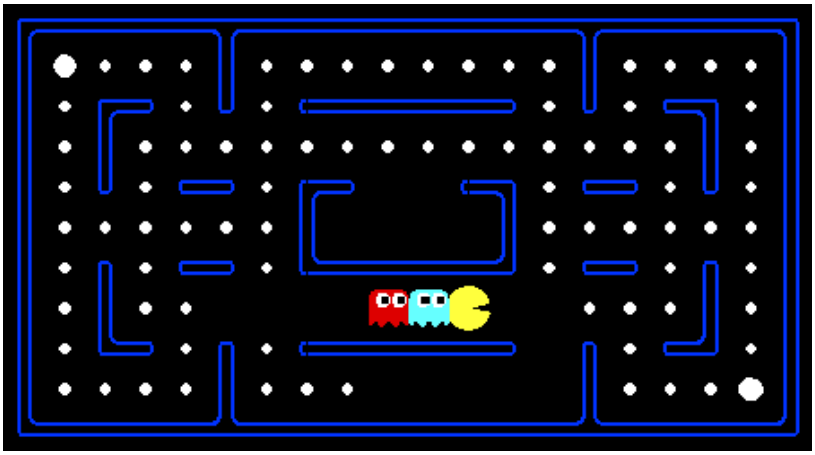
Proyecto 3: Clasificacion

Contenidos

- Introduccion
- Q1: Perceptron
- Q2: Analisis del Perceptron
- Q3: MIRA
- Q4: Clonando el Comportamiento del Pacman


Which Digit?


Which action?

Introduccion

En este proyecto, diseñará tres clasificadores: un clasificador perceptrón, un clasificador de gran margen (MIRA) y un clasificador perceptrón ligeramente modificado para la clonación conductual. Probará 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 ejecutará sus 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% (consulte LeNet-5 para ver un sistema de ejemplo en acción).

La clonación conductual 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.

El código lo podéis encontrar en el fichero zip que se os proporciona.

Ficheros con Datos	
`data.zip`	Contienen los datos sobre reconocimiento de dígitos
Ficheros que debeis editar	
`perceptron.py`	The location where you will write your perceptron classifier.
`mira.py`	The location where you will write your MIRA classifier.
`answers.py`	La respuesta a la 2. pregunta ira aquí.
Ficheros que no deberías editar	
`classificationMethod.py`	Super clase Abstracta para los clasificadores que vas a emplear.
`samples.py`	I/O codigo 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: Emplea el autograder para evaluar tu código. No cambies los nombres de ninguna clase ni método de los que se te proponen.

Pregunta 1 (5 puntos): Perceptron

Se es provee de un esquena básico que encontraréis en el `perceptron.py`. Aquí deberíais de rellenar la función `train`.

Dada la lista de rasgos f, el perceptrón computa (predice) la clase y' en base a la multiplicación escalar $f\,w$. Formalmente, dado el vector de rasgos f el score obtenido para cada clase será (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).

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

Después se seleccionará la clase y'' que presente un mayor valor *score* (producto dot (escalar)).

Ajustando los pesos

En el perceptron 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*:

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

Se compara y'con la verdadera clase 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. Eso implica que wy'' donde y'' *es* y debería haber obtenido una puntuación f más alta, y por el contrario wy' 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 consecuentemente:

$wy = wy + f$

$wy' = wy' - f$

Nota: Podeís emplear la suma, subtraction, y multiplication ed la clase `Counter` en `util.py`, o implementarla vosotros.

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 preoblemas del perceptrón es que es muy sensible a por ejemplo 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 los rasgos..... El presente código está configurado para realizar 3 iteraciones. Podéis modificar el número de iteraciones a través de la opción `-i iterations`. Empla diferentes números de iteraciones y comprueba como verían 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.

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 los rasgos más significativos (los que mayor peso presentan) para una determinada clase.

Pregunta

Rellena `findHighWeightFeatures(self, label)` en `perceptron.py`. Debería debolver las lista de los 100 rasgos con mayor peso para una determinada clase. Se imprimirán los 100 pixels (rasgos) con mayor peso ejecutando la siguiente llamada:

```
python dataClassifier.py -c perceptron -w
```

Emplealo y responde a la siguiente pregunta. ¿Cúal de las siguientes secuencias de pesos representa mejor lo aprendido por el perceptrón?

a

b

Responde a esta pregunta en el método q2 de `answers.py` `q2`, devolviendo una 'a' or 'b'.

Pregunta 3 (6 puntos): MIRA

Se provee del esqueleto básico para implementar MIRA en `mira.py`. MIRA es un clasificador online que se asemeja mucho a los vectores de soporte (support vector machine, SVM) y al perceptron. Rellena la función `trainAndTune (Para ello revisa las ptransparencias de la teoría)`.

`trainAndTune` in `mira.py`. Este método empleará los diferentes valores de capado *C* en `Cgrid`. Evaluate accuracy on the held-out validation set for each *C* and choose the *C* with the highest validation accuracy. In case of ties, prefer the *lowest* value of *C*. Test your MIRA implementation with:

```
python dataClassifier.py -c mira --autotune
```

Observationes:

- 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 el clasificador.
- Para usar un valor fijo de C = 0.001, elimine la opción --autotune del comando anterior.
- La tasa de acierto de validación y prueba cuando se usa --autotune debe estar alrededor del 60%.
- Se podría ahorrar algo de tiempo de depuración si el término +1 anterior se implementa como +1.0, debido al truncamiento por división de argumentos enteros. Dependiendo de cómo implemente esto, puede que no importe.

Pregunta 4 (4 puntos): Clonación de comportamiento

Has construido dos tipos diferentes de clasificadores, un clasificador perceptrón y mira. Ahora usará una versión modificada de perceptron para aprender de los agentes pacman. En esta pregunta, completará los métodos de clasificación y capacitación en `perceptron_pacman.py`. Este código debe ser similar a los métodos 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, y las características extraídas son una función tanto del estado como de la posible etiqueta.

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

$$score(s,a)=w*f(s,a)$$

Luego, el clasificador asigna cualquier etiqueta que reciba la puntuación más alta:

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

Las actualizaciones de capacitación 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:

$$w=w+f(s,a)$$

la acción correcta

$w = w - f(s, a')$

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%.

Pregunta Adicional (6 puntos adicionales): Diseño de los rasgos de los dígitos

Crear clasificadores es solo una pequeña parte para lograr que un buen sistema funcione para una tarea. De hecho, la principal diferencia entre un buen sistema de clasificación y uno malo generalmente no es el clasificador en sí (por ejemplo, perceptrón vs. Bayes ingenuo), sino la calidad de las características utilizadas. Hasta ahora, hemos utilizado las características más simples posibles: la identidad de cada píxel (estar activado / desactivado).

Para aumentar aún más la precisión de su clasificador, deberá extraer características más útiles de los datos. EnhancedFeatureExtractorDigit en dataClassifier.py es su nuevo espacio para experimentar. Al analizar los resultados de sus clasificadores, se observan algunos algunos errores y se ha de buscar características de la entrada que le darían al clasificador información adicional útil sobre la etiqueta. Se puede agregar código a la función de análisis en dataClassifier.py para inspeccionar lo que está haciendo su clasificador. Por ejemplo, en los datos de dígitos, es conveniente considerar el número de regiones separadas y conectadas de píxeles blancos, que varía según el tipo de dígito. 1, 2, 3, 5, 7 tienden a tener una única (o dos) región contigua de espacios en blanco, mientras que los dígitos 6, 8, 9 crean más. El número de regiones blancas en un 4 depende del escritor. Este es un ejemplo de una característica que no está directamente disponible para el clasificador a partir de la información por píxel. Si su extractor de características agrega nuevas características que codifican estas propiedades, el clasificador podrá explotarlas. Hay que tener en cuenta que algunas características pueden requerir un cálculo no trivial para extraer, por lo tanto, procura escribir un código eficiente y correcto.

Nota: se trabajará con dígitos, así que asegúrese de usar DIGIT_DATUM_WIDTH y DIGIT_DATUM_HEIGHT, en lugar de FACE_DATUM_WIDTH y FACE_DATUM_HEIGHT.

Pregunta

Agregar nuevas características binarias para el conjunto de datos de dígitos en la función EnhancedFeatureExtractorDigit. Tenga en cuenta que puede codificar una característica que toma 3 valores [1,2,3] utilizando 3 características binarias, de las cuales solo una está activada en ese momento, para indicar cuál de las tres posibilidades tiene. Para probar el clasificador se empleará el siguiente comando:

python dataClassifier.py -d dígitos -c naiveBayes -f -a -t 1000

Con las características básicas (sin la opción -f), la elección óptima del parámetro de suavizado debería producir un 82% en el conjunto de validación con un rendimiento de prueba del 78%. Se recibirán 3 puntos por implementar nuevas características que produzcan cualquier mejora. Se recibirán 3 puntos adicionales si las nuevas características le ofrecen un rendimiento de prueba mayor o igual al 84% con el comando anterior.