Universidad del País Vasco

Euskal Herriko Unibertsitatea

Técnicas de Inteligencia Artificial

Ingeniería Informática de Gestión y Sistemas de Información

Practica 3
Clasificacion

Autor(es):

Xabier Gabiña
Diego Montoya

23 de noviembre de 2024

Índice general

Índice de Códigos

1. Introducción

En este proyecto, se abordará la implementación de algoritmos de clasificación utilizando el modelo de perceptrón, enfocado en resolver problemas de lógica y clonación de comportamiento en un entorno de aprendizaje. El desarrollo comienza con la construcción de puertas lógicas básicas (AND, OR, NOT y XOR) mediante perceptrones para comprender el funcionamiento del modelo en conjuntos de datos limitados. Posteriormente, se extiende su aplicación hacia la creación de clasificadores más complejos capaces de reconocer dígitos y replicar comportamientos observados en un agente de juego.

2. Ejercicios

2.1. Puertas Lógicas con perceptron

Descripción

En este apartado se implementaran las puertas lógicas AND, OR, NOT y XOR mediante perceptrones. Empezaremos haciendo pruebas ya conociendo los pesos para pasar a entrenar nosotros mismos los pesos y para finalizar utilizaremos el weighted average.

Implementación

```
# Definir dos vectores (listas): input my_x, pesos my_w
my_x = [0, 1]#input un item
my_w = [0.66, 0.80]

# Multiplicar dos vectores elemento a elemento
def mul(a, b):
    """
    Devolver una lista c, de la misma longitud que a y b, donde
    cada elemento c[i] = a[i] * b[i]
    """
    return [a[i] * b[i] for i in range(len(a))]

mul(my_x, my_w)

my_bias  = 1
my_wbias = -0.97

my_wPlusWBias = [my_wbias] + my_w

def distanciaDelCoseno(x, weights, bias):
    """
    El producto escalar (producto punto) de dos vectores y la similitud de coseno no son
    completamente equivalentes
    ya que la similitud del coseno solo se preocupa por la diferencia de ángulo,
    mientras que el producto de punto se preocupa por el ángulo y la magnitud
    Pero en muchas ocasiones se emplean indistintamente
    Así pues, esta función devuelve el valor escalar de la neurona, es decir,
    el producto escalar entre el vector de entrada añadiendo el bias y el vector de los pesos
    recordad que "sum(list)" computa la suma de los elementos de una lista
    Así pues se comenzará por añadir el bías en la posición 0 del vector de entrada
    antes de llevar a cabo el producto escalar para así tener dos vectores de
    la misma longitud. Emplea la función mul que ya has programado
    """
    x = [bias] + x
    return sum(mul(x, weights))

distanciaDelCoseno(my_x, my_wPlusWBias, my_bias)

def neuron(x, weights, bias):
    """
    Devolverá el output de una neurona clásica
    (reutilizar la distancia del coseno definida previamente)
    y añadir la función de activación (step function): si >=0 entonces 1 sino -1
    """
    return 1 if distanciaDelCoseno(x, weights, bias)>=0 else -1

neuron(my_x, my_wPlusWBias, my_bias)

def and_neuron(x):
    """
    Devuelve x1 AND x2 suponiendo que la hemos entrenado
    y que en ese entrenamiento hemos aprendido los pesos apropiados
    (mirar las transparencias de clase). Así pues inicializaremos
    una la variable local and_w con los pesos aprendidos
    y a 1 la variable local and_bias
```

```
    y ejecutaremos la función neurona para el item x"""
    and_w    = [-0.97,0.66, 0.80]#initialization of the weights and_w
    and_bias = 1#initialization of the bias and_bias
    return neuron(x, and_w, and_bias)

my_x_collection = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1],
]

print('Testando el output de la neurona AND')
#bucle para ir obteniendo el output de la neurona AND para cada item del input
for my_x in my_x_collection:
    print(my_x, f'{and_neuron(my_x):.3f}')

```

Código 2.1: Implementación del perceptron para las puerta lógica AND

```python
from random import seed, random

# Inicialización
print('Entrenando una neurona OR hasta convergencia')
notConverge=True
seed(1)

orWeights= [random() for i in range(3)]
orBias = 1
orGoldOutputs = [-1,1,1,1]

# Entrenamiento
numeroVuelta = 0
while notConverge:
    notConverge = False
    for i, my_x in enumerate(my_x_collection):
        x_with_bias = [orBias] + my_x  # Agrega el bias al vector de entrada
        predicted_output = neuron(my_x, orWeights, orBias)

        # Si la predicción no coincide con la salida esperada, ajustar pesos
        if predicted_output != orGoldOutputs[i]:
            adjustment = 1 if orGoldOutputs[i] == 1 else -1
            orWeights = [orWeights[j] + adjustment * x_with_bias[j] for j in range(len(orWeights))
    ]
            notConverge = True  # Continuar iterando hasta que todo esté correcto
    numeroVuelta += 1
    print(f"Vuelta {numeroVuelta}: Pesos actualizados OR:", orWeights)

```

Código 2.2: Implementación del perceptron para las puerta lógica OR

```python
my_x_collection = [
    [0],
    [1]
]
from random import seed, random


# Inicializaciones
print('Entrenando una neurona NOT hasta convergencia')
notConverge = True
seed(1)

notWeights = [random(), random()]
notBias    = 1
notGoldOutput = [1, -1]

#entrenando
numeroVuelta = 0
while notConverge:
    notConverge = False
    for i, my_x in enumerate(my_x_collection):
        x_with_bias = [orBias] + my_x  # Agrega el bias al vector de entrada
        predicted_output = neuron(my_x, orWeights, orBias)

        # Si la predicción no coincide con la salida esperada, ajustar pesos
        if predicted_output != orGoldOutputs[i]:
            adjustment = 1 if orGoldOutputs[i] == 1 else -1
            orWeights = [orWeights[j] + adjustment * x_with_bias[j] for j in range(len(orWeights))
    ]
            notConverge = True  # Continuar iterando hasta que todo esté correcto
    numeroVuelta += 1
    print(f"Vuelta {numeroVuelta}: Pesos actualizados NOT:", orWeights)


```

Código 2.3: Implementación del perceptron para las puerta lógica NOT

```
my_x_collection = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1],
]

def matrixAverage(m):
  res=list()
  acum=list()
  if len(m) > 0:
      res=[0]*len(m[0])
      for v in m:
          res = [a+b for a,b in zip (res,v)]
      acum=[elem/len(m) for elem in res]
  return acum

matrix=[[2,3,4],[2,3,4],[2,3,4]]
print(matrixAverage(matrix))

from random import seed, random

# Inicializaciones
print('Entrenando una neurona OR hasta convergencia')
notConverge=True
seed(1)

orWeights= [random() for i in range(3)]
orBias = 1
orGoldOutputs = [-1,1,1,1]

# Historial de pesos
weightHistory = []

# Entrenamiento
numeroVuelta = 0
while notConverge:
    notConverge = False
    epoch_weights = orWeights.copy()  # Copia de los pesos para acumular en cada epoch
    for i, my_x in enumerate(my_x_collection):
        x_with_bias = [orBias] + my_x  # Agrega el bias al vector de entrada

        # Verificar si la predicción coincide con la salida esperada
        if neuron(my_x, orWeights, orBias) != orGoldOutputs[i]:
            # Ajuste de pesos basado en el error
            adjustment = 1 if orGoldOutputs[i] == 1 else -1
            orWeights = [orWeights[j] + adjustment * x_with_bias[j] for j in range(len(orWeights))
    ]
            notConverge = True  # Continuar iterando hasta convergencia
    weightHistory.append(orWeights.copy())  # Guardar los pesos de esta epoch
    numeroVuelta += 1
    print(f"Vuelta {numeroVuelta}: Pesos actualizados OR:", orWeights)

# Calcular el promedio ponderado de los pesos al finalizar el entrenamiento
averageWeights = matrixAverage(weightHistory)
print("\nPromedio ponderado de los pesos OR:", averageWeights)

def or_neuron(x):
    """
    Devuelve x1 AND x2 suponiendo que la hemos entrenado
    y que en ese entrenamiento hemos aprendido los pesos apropiados
    (mirar las transparencias de clase). Así pues inicializaremos
    una la variable local and_w con los pesos aprendidos
    y a 1 la variable local and_bias
    y ejecutaremos la función neurona para el item x"""
    or_w    = [-0.3656,0.8474, 0.7637]#initialization of the weights and_w
    or_bias = 1#initialization of the bias and_bias
    return neuron(x, or_w, or_bias)

print('Testando el output de la neurona OR')
#bucle para ir obteniendo el output de la neurona AND para cada item del input
```

```
for my_x in my_x_collection:
    print(f'Input: {my_x} -> OR Output: {or_neuron(my_x):.3f}')

```

Código 2.4: Implementación del perceptron para las puerta lógica OR con Weighted average

```python
# Combinando una puerta OR y una AND, y aprendiendo el peso que hay que darle a cada una para obtener un XOR
from random import seed, random

# Inicializaciones
print('Entrenando una neurona XOR hasta convergencia')
xorConverge=True
seed(1)

xorWeights= [random() for i in range(3)]
xorBias   = -0.5
xorGoldOutputs=[1,-1,-1,1]

# Entrenando
numeroVuelta = 0
while xorConverge:
    xorConverge = False
    for i, my_x in enumerate(my_x_collection):
        # Usar las salidas de las puertas AND y OR como entradas para la XOR
        and_output = and_neuron(my_x)
        or_output = or_neuron(my_x)

        # Crear el nuevo vector de entrada con los resultados de AND y OR
        new_x = [xorBias, and_output, or_output]

        # Verificar si la predicción coincide con el valor esperado
        if neuron([and_output, or_output], xorWeights, xorBias) != xorGoldOutputs[i]:
            # Ajuste de pesos
            adjustment = 1 if xorGoldOutputs[i] == 1 else -1
            xorWeights = [xorWeights[j] + adjustment * new_x[j] for j in range(len(xorWeights))]
            xorConverge = True  # Continuar iterando hasta convergencia
    numeroVuelta += 1
    print(f"Vuelta {numeroVuelta}: Pesos actualizados XOR:", xorWeights)

def xor_neuron(x):
  """
  Return x1_ * x2 + x1 * x2_
  """
  xor_w    = [-1.115635755887599, 0.3474337369372327, -0.7362253810233859]
  xor_bias = -0.5
  new_x=list()
  new_x.append(and_neuron(x))
  new_x.append(or_neuron(x))
  return neuron(new_x, xor_w, xor_bias)

print('Checking XOR neuron output')
for my_x in my_x_collection:
    print(my_x, f'{xor_neuron(my_x):.3f}')

```

Código 2.5: Implementación del perceptron para las puerta lógica XOR

Conclusiones

Respecto a la implementación de AND ha sido sencilla ya que lo más complicado que es encontrar los pesos no hemos tenido que realizarlo por lo que no hay mucho que comentar al respecto.

En cuanto a la de OR y NOT de a continuación, ya hemos tenido que realizar el entrenamiento de los pesos. Aqui hemos tenido algun problema con el tema de añadir el BIAS y la longitud de los vectores, pero una vez solucionado hemos podido realizar el entrenamiento de los pesos sin mayor problema.

En cuanto al weighted average, es practicamente el mismo entrenamiento pero guardando los pesos en cada iteración para luego calcular la media ponderada de los pesos.

Por último, la implementación de XOR ha sido la más complicada ya que hemos tenido que combinar las puertas OR y AND para obtener el resultado de la XOR. Una vez obtenido el resultado de las puertas OR y AND, hemos tenido que realizar el entrenamiento de los pesos para obtener el resultado de la XOR pero si nos ha costado ver como podiamos combinar las puertas OR y AND para obtener el resultado de la XOR.

2.2. Perceptron

Descripción

El ejercicio consiste en implementar una función de entrenamiento para un modelo de perceptrón, con el objetivo de clasificar digitos en diferentes categorías. Este modelo de perceptrón es un tipo de algoritmo de aprendizaje supervisado que ajusta un vector de pesos en función de los errores de clasificación en el conjunto de entrenamiento.

El perceptrón funciona de la siguiente manera:

1. Obtenemos los datos y los pesos asociados a cada etiqueta.
2. Iteramos sobre los datos de entrenamiento y calculamos los puntajes para cada etiqueta. Este puntuaje se consigue multiplicando el vector de características por el vector de pesos de cada etiqueta.
3. Predecimos la etiqueta con el puntaje más alto.
4. Si la predicción es incorrecta, actualizamos los pesos de la etiqueta correcta y de la etiqueta predicha.
5. Repetimos el proceso hasta que se alcance el número máximo de iteraciones o hasta que no haya errores de clasificación.

Implementación Final

```
class PerceptronClassifier:
  """
  Perceptron classifier.

  Note that the variable 'datum' in this code refers to a counter of features
  (not to a raw samples.Datum).
  """

  def __init__(self, legalLabels, max_iterations):
      self.legalLabels = legalLabels
      self.type = "perceptron"
      self.max_iterations = max_iterations
      self.weights = {}
      self.features = None
      for label in legalLabels:
          self.weights[label] = util.Counter()  # this is the data-structure you should use

  def setWeights(self, weights):
      assert len(weights) == len(self.legalLabels)
      self.weights = weights

  def train(self, trainingData, trainingLabels, validationData, validationLabels):
      """
      The training loop for the perceptron passes through the training data several
      times and updates the weight vector for each label based on classification errors.
      See the project description for details.

      Use the provided self.weights[label] data structure so that
      the classify method works correctly. Also, recall that a
      datum is a counter from features to values for those features
      (and thus represents a vector a values).
      """

      self.features = trainingData[0].keys()  # could be useful later
      # DO NOT ZERO OUT YOUR WEIGHTS BEFORE STARTING TRAINING, OR
      # THE AUTOGRADER WILL LIKELY DEDUCT POINTS.

      for iteration in range(self.max_iterations):
          print("Starting iteration ", iteration, "...")
          for i in range(len(trainingData)):  # training data
              # Obtener el ejemplo y su etiqueta real
              x_i = trainingData[i]
              y_i = trainingLabels[i]
```

```
44
45              # Calcular los puntajes para cada etiqueta
46              scores = util.Counter()
47              for label in self.legalLabels:
48                  scores[label] = self.weights[label] * x_i
49
50              # Predecir la etiqueta con el puntaje más alto
51              predicted_label = scores.argMax()
52
53              # Si la predicción es incorrecta, actualizar los pesos
54              if predicted_label != y_i:
55                  self.weights[y_i] += x_i
56                  self.weights[predicted_label] -= x_i
57
58  def classify(self, data):
59      """
60      Classifies each datum as the label that most closely matches the prototype vector
61      for that label.  See the project description for details.
62
63      Recall that a datum is a util.counter...
64      """
65      guesses = []
66      for datum in data:
67          vectors = util.Counter()
68          for label in self.legalLabels:
69              vectors[label] = self.weights[label] * datum
70          guesses.append(vectors.argMax())
71      return guesses
72
73  def findHighWeightFeatures(self, label):
74      """
75      Returns a list of the 100 features with the greatest weight for some label
76      """
77      # Obtener los pesos de los features para la etiqueta dada
78      weights = self.weights[label]
79
80      # Ordenar los features por peso en orden descendente
81      sorted_features = weights.sortedKeys()
82
83      # Seleccionar los 100 features con mayor peso
84      featuresWeights = sorted_features[:100]
85
86      return featuresWeights
87
```

Código 2.6: Implementación final del perceptron

Conclusiones

La implementación del perceptron es bastante diferente a lo que hemos empezado haciendo en el ejercicio de las puertas lógicas aunque la idea es la misma. La diferencia es que ya no tenemos una clasificación binaria sino que tenemos varias etiquetas y por lo tanto tenemos que calcular los puntajes para cada etiqueta y predecir la etiqueta con el puntaje más alto. Esto, tambien provoca que las actualizaciones de los pesos sean diferentes ya que ahora tenemos que actualizar los pesos de la etiqueta correcta y de la etiqueta predicha. De esta forma, sumando a la etiqueta correcta y restando a la etiqueta predicha conseguimos que la etiqueta correcta se vuelve cada vez más probable en comparación con las etiquetas incorrectas.

2.3. Clonando el Comportamiento del Pacman

Descripción

En este ejercicio se implementará un clonador de comportamiento para el agente Pacman, utilizando un modelo de perceptrón para clasificar los movimientos legales en un estado de juego. El objetivo es entrenar el clonador para que pueda predecir el movimiento correcto del Pacman en función de las características de un estado de juego.

Primera Implementación

```
class PerceptronClassifierPacman(PerceptronClassifier):
    def __init__(self, legalLabels, maxIterations):
        PerceptronClassifier.__init__(self, legalLabels, maxIterations)
        self.weights = util.Counter()

    def classify(self, data):
        """
        Data contains a list of (datum, legal moves)

        Datum is a Counter representing the features of each GameState.
        legalMoves is a list of legal moves for that GameState.
        """
        guesses = []
        for datum, legalMoves in data:
            vectors = util.Counter()
            for move in legalMoves:
                vectors[move] = self.weights * datum[move]  # changed from datum to datum[1]
            guesses.append(vectors.argMax())
        return guesses

    def train(self, trainingData, trainingLabels, validationData, validationLabels):
        self.features = trainingData[0][0]['Stop'].keys()  # could be useful later
        # DO NOT ZERO OUT YOUR WEIGHTS BEFORE STARTING TRAINING, OR
        # THE AUTOGRADER WILL LIKELY DEDUCT POINTS.

        for iteration in range(self.max_iterations):
            print("Starting iteration ", iteration, "...")
            for i in range(len(trainingData)):
                #print(trainingData[i])
                # Obtener el ejemplo, el movimiento legal y la etiqueta
                x_i, legalMoves = trainingData[i]
                y_i = trainingLabels[i]

                # Calcular los puntajes para cada etiqueta
                scores = util.Counter()
                for move in legalMoves:
                    #print(x_i[move])
                    scores[move] = self.weights[move] * x_i[move]['foodCount']

                # Obtener la etiqueta con el puntaje más alto
                predicted_move = scores.argMax()

                # Si la predicción es incorrecta, actualizar los pesos
                if y_i != predicted_move:
                    self.weights[y_i] += x_i[predicted_move]['foodCount']
                    self.weights[predicted_move] -= x_i[predicted_move]['foodCount']

```

Código 2.7: Implementación inicial del clonador de comportamiento del pacman

Final Implementación

```python
class PerceptronClassifierPacman(PerceptronClassifier):
    def __init__(self, legalLabels, maxIterations):
        PerceptronClassifier.__init__(self, legalLabels, maxIterations)
        self.weights = util.Counter()

    def classify(self, data):
        """
        Data contains a list of (datum, legal moves)

        Datum is a Counter representing the features of each GameState.
        legalMoves is a list of legal moves for that GameState.
        """
        guesses = []
        for datum, legalMoves in data:
            vectors = util.Counter()
            for move in legalMoves:
                vectors[move] = self.weights * datum[move]  # changed from datum to datum[1]
            guesses.append(vectors.argMax())
        return guesses

    def train(self, trainingData, trainingLabels, validationData, validationLabels):
        self.features = trainingData[0][0]['Stop'].keys()  # could be useful later
        # DO NOT ZERO OUT YOUR WEIGHTS BEFORE STARTING TRAINING, OR
        # THE AUTOGRADER WILL LIKELY DEDUCT POINTS.

        for iteration in range(self.max_iterations):
            print("Starting iteration ", iteration, "...")
            for i in range(len(trainingData)):
                # Obtener el ejemplo, el movimiento legal y la etiqueta
                x_i, legalMoves = trainingData[i]
                y_i = trainingLabels[i]

                # Calcular los puntajes para cada etiqueta
                scores = util.Counter()
                for move in legalMoves:
                    scores[move] = self.weights * x_i[move]

                # Obtener la etiqueta con el puntaje más alto
                predicted_move = scores.argMax()

                # Si la predicción es incorrecta, actualizar los pesos
                if y_i != predicted_move:
                    self.weights += x_i[y_i]
                    self.weights -= x_i[predicted_move]

```

Código 2.8: Implementación final del clonador de comportamiento del pacman

Conclusiones

En este ejercicio nos hemos hecho un poco lio con las estructuras, especialmente, con las del peso (Por no leer bien el enunciado para variar). Lo principal es que ahora trainingData es una lista con dos elementos, el primero es un diccionario con los movimientos y sus valores y el segundo es una lista con los movimientos legales. Tambien el self.weights es un vector de pesos unicos en vez de un diccionario ya que ahora estas con las clasificación de decisiones por lo que el vector de pesos es compartido mientras que antes era un diccionario con los pesos de cada etiqueta.

Teniendo en cuenta esto, la implementación es bastante similiar a la del Perceptron del ejercicio anterior.

2.4. Clonando el Comportamientodel Pacman con rasgos diseñados por nosotros

Descripción

En este ejercicio tendremos que definir nuestros propios rasgos para el clonador de comportamiento del Pacman y entrenar el modelo de perceptrón con estos rasgos. En este caso, el ejercicio ya se nos da implementado por lo que solamente explicaremos en el apartado de Conclusiones el porqué de los rasgos elegidos.

Conclusiones

Las features que se han elegido son:

- FoodDistance: Distancia al punto más cercano de la comida.
- GhostDistance: Distancia al punto más cercano de un fantasma.

Food Distance es una feature basica ya que el propio objetivo del pacman es comerse la comida por lo que es importante saber la distancia a la comida más cercana. Para ello la funcion recorre todas las posiciones de comida (foods) y calcula la distancia de Manhattan entre la posición de Pac-Man (pac) y cada posición de comida. De todas estas distancias, se selecciona la menor (minD), la comida más cercana.

Ghost Distance de forma inversamente proporcional a la Food Distance es una feature que nos permite seguir vivos ya que si nos acercamos a un fantasma, este nos matará. Por lo que es importante saber la distancia a los fantasmas para evitarlos. Para ello la función recorre las posiciones de los fantasmas (ghostPositions) y calcula la distancia de Manhattan entre Pac-Man y cada fantasma. La distancia mínima (minD) se guarda como la distancia al fantasma más cercano.

Aunque los resultados de estas dos metricas no son malos, la realidad es que no llegamos al esperado 90 % de aciertos como nos dice el enunciado. Esto podria ser solventado añadiendo más features que nos permitan tener una mejor clasificación.

3. Resultados

3.1. Casos de prueba

3.1.1. Perceptron

```
python dataClassifier.py -c perceptron
Doing classification
--------------------
data:           digits
classifier:             perceptron
using enhanced features?:       False
training set size:      100
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Validating...
55 correct out of 100 (55.0%).
Testing...
48 correct out of 100 (48.0%).

python dataClassifier.py -c perceptron -i 9
Doing classification
--------------------
data:           digits
classifier:             perceptron
using enhanced features?:       False
training set size:      100
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
Starting iteration  6 ...
Starting iteration  7 ...
Starting iteration  8 ...
Validating...
56 correct out of 100 (56.0%).
Testing...
54 correct out of 100 (54.0%).

```

Código 3.1: Ejecución del perceptron

3.1.2. Clonando el Comportamiento del Pacman

```
python dataClassifier.py -c perceptron -d pacman
Doing classification
--------------------
data:           pacman
classifier:             perceptron
using enhanced features?:       False
training set size:      100
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Validating...
83 correct out of 100 (83.0%).
Testing...
```

```
80 correct out of 100 (80.0%).

python dataClassifier.py -c perceptron -d pacman -i 9
Doing classification
--------------------
data:           pacman
classifier:             perceptron
using enhanced features?:       False
training set size:      100
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
Starting iteration  6 ...
Starting iteration  7 ...
Starting iteration  8 ...
Validating...
83 correct out of 100 (83.0%).
Testing...
80 correct out of 100 (80.0%).

```

Código 3.2: Ejecución del clonador de comportamiento del pacman

3.1.3. Clonando el Comportamiento del Pacman con rasgos diseñados por nosotros

```
python dataClassifier.py -c perceptron -d pacman -f -g ContestAgent -t 1000 -s 1000
Doing classification
--------------------
data:           pacman
classifier:             perceptron
using enhanced features?:       True
training set size:      1000
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Validating...
812 correct out of 1000 (81.2%).
Testing...
832 correct out of 1000 (83.2%).

python dataClassifier.py -c perceptron -d pacman -f -g StopAgent -t 1000 -s 1000
Doing classification
--------------------
data:           pacman
classifier:             perceptron
using enhanced features?:       True
training set size:      1000
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Validating...
502 correct out of 502 (100.0%).
Testing...
805 correct out of 805 (100.0%).

python dataClassifier.py -c perceptron -d pacman -f -g FoodAgent -t 1000 -s 1000
Doing classification
--------------------
data:           pacman
classifier:             perceptron
```

```
using enhanced features?:       True
training set size:      1000
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Validating...
285 correct out of 346 (82.4%).
Testing...
307 correct out of 380 (80.8%).

python dataClassifier.py -c perceptron -d pacman -f -g SuicideAgent -t 1000 -s 1000
Doing classification
--------------------
data:           pacman
classifier:             perceptron
using enhanced features?:       True
training set size:      1000
Extracting features...
Training...
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Validating...
269 correct out of 339 (79.4%).
Testing...
87 correct out of 102 (85.3%).

```

Código 3.3: Ejecución del clonador de comportamiento del pacman con rasgos diseñados por nosotros

3.2. Autograder

```
python autograder.py
Extracting features...
Extracting features...
Starting on 11-8 at 18:30:14

Question q2
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
79 correct out of 100 (79.0%).
*** PASS: test_cases/q2/grade.test (4 of 4 points)
***     79.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points

### Question q2: 4/4 ###


Question q3
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
80 correct out of 100 (80.0%).
*** PASS: test_cases/q3/contest.test (2 of 2 points)
***     80.0 correct (2 of 2 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  2 points
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
72 correct out of 100 (72.0%).
*** PASS: test_cases/q3/suicide.test (2 of 2 points)
***     72.0 correct (2 of 2 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  2 points

### Question q3: 4/4 ###


Question q4
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
95 correct out of 100 (95.0%).
*** PASS: test_cases/q4/contest.test (2 of 2 points)
***     95.0 correct (2 of 2 points)
***         Grading scheme:
***          < 90:  0 points
***         >= 90:  2 points
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
```

```
85 correct out of 100 (85.0%).
*** PASS: test_cases/q4/suicide.test (2 of 2 points)
***     85.0 correct (2 of 2 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  2 points

### Question q4: 4/4 ###


Finished at 18:30:29

Provisional grades
==================
Question q2: 4/4
Question q3: 4/4
Question q4: 4/4
------------------
Total: 12/12

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

```

Código 3.4: Ejecución del autograder