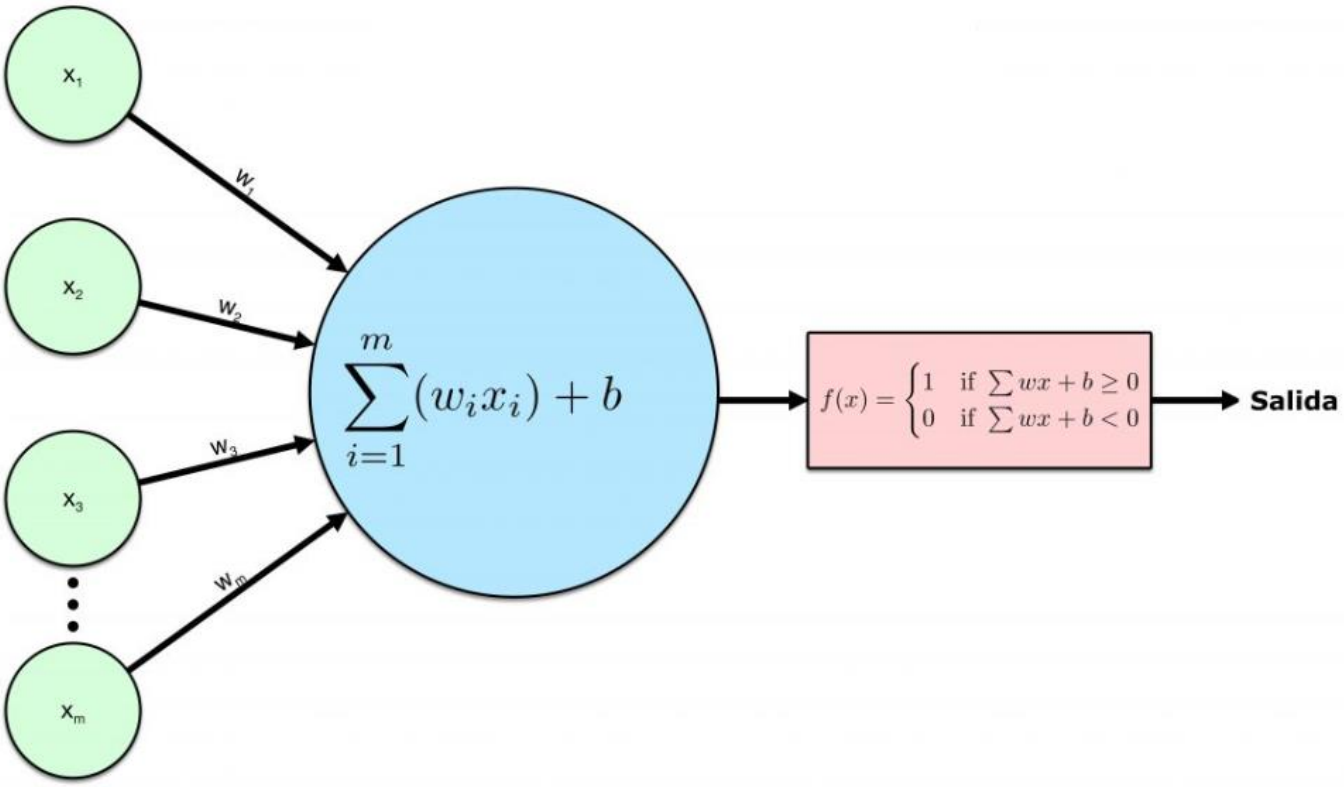
Técnicas de Inteligencia Artificial

Perceptrones en redes neuronales



Realizado por:

Cuesta Alario, David

Fernández Andrés, Cristian

Índice

Reconocimiento óptico de caracteres

Vamos a entrenar un perceptron para que aprenda a identificar una colección de dígitos numéricos escritos a mano. Dichos datos han sido obtenidos mediante una imagen que ha sido reprocesada en forma de texto

Perceptron simple

```
class PerceptronClassifier:

    def __init__( self, legalLabels, max_iterations):
        self.legalLabels = legalLabels
        self.type = "perceptron"
        self.max_iterations = 5
        self.weights = {}
        for label in legalLabels:
            self.weights[label] = util.Counter()
```

La clase **PerceptronClassifier** dispone de los siguientes atributos:

- **LegalLabels** se trata de una lista que contendrá todos los valores posibles que puede contener cada etiqueta. Se pasara por parámetro en la constructora de la clase.

  max_iterations se trata de un número entero que representa el número máximo de veces que se va a entrenar el perceptron con los datos de entrenamiento en caso de que no converja. Se puede pasar por parámetro, pero en nuestro caso lo hemos preestablecido a cinco dado que obtiene un modelo suficientemente preciso para nuestros datos

- **weights** se trata de un diccionario que contendrá la matriz de pesos del perceptron. Este es el dato que se pretende entrenar

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

La función **setWeights** permite inicializar la matriz de pesos a través de un parámetro

```
def train( self, trainingData, trainingLabels, validationData, validationLabels ):

    self.features = trainingData[0].keys()

    for iteration in range(self.max_iterations):
        print "Starting iteration ", iteration, "..."

        for i in range(len(trainingData)):
            MaxEstimacion = None
            MaxLabel = None

            for label in self.legalLabels:
                estimacion = trainingData[i] * self.weights[label]

                if ( MaxEstimacion is None or estimacion > MaxEstimacion ):
                    MaxEstimacion = estimacion
                    Maxlabel = label

            claseReal = trainingLabels[i]
            claseEstimada = Maxlabel
            if( claseReal != claseEstimada  ):
                self.weights[claseReal]     = self.weights[claseReal] + trainingData[i]
                self.weights[claseEstimada] = self.weights[claseEstimada] - trainingData[i]
...
```

La función **train** realiza el entrenamiento del perceptron

- Requerirá que se le pasen por parámetro los siguientes datos:
  - **trainingData** y **validationData** se trata de dos diccionarios que contendrán respectivamente:
    - Una matriz con todos los datos que se pretende utilizar para llevar cabo el entrenamiento del modelo.
    - Una matriz con los datos prototipo que utilizamos como modelo para las predicciones

    La clave del diccionario son las coordenadas del pixel y el valor es un booleano que indica si está pintado o no

    ```
    (13, 16): 0, (2, 23): 0, (26, 8): 0, (0, 11): 0, (3, 6): 0, (27, 5): 0,
    ```

  - **trainingLabels** y **validationLabels** se trata de dos listas que contienen respectivamente:
    - La clase real de cada uno de los datos de entrenamiento
    - La clase real de los datos prototipo

    ```
    [5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9, 4, 0, 9, 1, 1,
    ```

- Se utilizaran las siguientes variables locales
  - **iteration** representa el número de veces que hemos recorrido todos los casos de entrenamiento
  - **i** representa la posición de cada una de las instancias en la matriz de entrenamiento
  - **label** representa cada una de las clases permitidas que queremos que nuestro modelo aprenda a predecir
  - **estimación** se trata del valor estimado con el que puntúa nuestro modelo a cada una de las clases permitidas
  - **MaxEstimacion** se trata del valor estimado de mayor valor
  - **MaxLabel** se trata de la clase permitida que ha obtenido la mayor puntuación
  - **ClaseReal** Es la clase que debería haber predicho nuestro modelo.
  - **ClaseEstimada** Es la clase que ha predicho nuestro modelo.
- Para completar el entrenamiento realizaremos las siguientes operaciones tantas veces como iteraciones se han dispuesto. (cinco en nuestro caso)
  - Por cada dato de entrenamiento
    - Reseteraremos los valores de **MaxEstimacion** y **MaxLabel** a None
    - Por cada clase permitida
      - Calcularemos el valor estimado por nuestro modelo como el producto del peso asignado a dicha clase permitida y el valor del dato de entrenamiento que estamos analizando
      - Nos guardaremos en las variables **MaxEstimacion** y **MaxLabel** los valores máximos de cada estimación y la clase a la que le corresponde dicha estimación
    - Establecemos que la **ClaseReal** es la que viene en los parámetros de entrenamiento para la instancia actual y la clase **ClaseEstimada** es la que ha obtenido la mayor puntuación por parte de nuestro modelo
    - Si la **ClaseReal** y la **ClaseEstimada** son la misma *"virgencita, virgencita que me quede como estoy"*. En caso contrario podemos asumir que nuestro modelo se ha equivocado en la predicción y procederemos a ajustar los pesos de la siguiente manera:
      - Actualizamos los pesos de la **ClaseReal** sumándole los valores del dato de entrenamiento que estamos analizando con la finalidad de favorecer en un futuro que se prediga esta clase si se obtiene un dato como de entrenamiento como el actual.
      - Actualizamos los pesos de la **ClaseEstimada** restándole los valores del dato de entrenamiento que estamos analizando con la finalidad de penalizar en un futuro que se prediga esta clase si se obtiene un dato como de entrenamiento como el actual.

```
def classify(self, data ):
    guesses = []

    for datum in data:
        vectors = util.Counter()

        for l in self.legalLabels:
            vectors[l] = self.weights[l] * datum

        guesses.append(vectors.argMax())

    return guesses
```

La función **classify** lleva a cabo el proceso completo de entrenamiento para múltiples neuronas clasificación

```
def findHighWeightFeatures(self, label):
    featuresWeights = []
    featuresWeights = self.weights[label].sortedKeys()[0:100]
    return featuresWeights
```

La función **findHighWeightFeatures** permite identifique los rasgos más significativos para una determinada clase obteniendo los que obtienen mayor peso.

Para obtenerlos simplemente ordenamos la matriz de pesos de mayor a menor mediante la función **shortedKeys**() de la librería **util. py** y nos quedamos con los 100 primeros

Perceptron mejorado con MIRA

```
class MiraClassifier:

    def __init__( self, legalLabels, max_iterations):
        self.legalLabels = legalLabels
        self.type = "mira"
        self.automaticTuning = False
        self.C = 0.001
        self.legalLabels = legalLabels
        self.max_iterations = max_iterations
        self.initializeWeightsToZero()

    def initializeWeightsToZero(self):
        self.weights = {}
        for label in self.legalLabels:
            self.weights[label] = util.Counter()

    def train(self, trainingData, trainingLabels, validationData, validationLabels):

        self.features = trainingData[0].keys()

        if (self.automaticTuning):
            Cgrid = [0.002, 0.004, 0.008]
        else:
            Cgrid = [self.C]

        return self.trainAndTune(trainingData, trainingLabels, validationData, validationLabels, Cgrid)
```

La clase **MiraClassifier** dispone de los siguientes atributos:

- **LegalLabels** se trata de una lista que contendrá todos los valores posibles que puede contener cada etiqueta.
  Se pasara por parámetro en la constructora de la clase.
- **max_iterations** se trata de un número entero que representa el número máximo de veces que se va a entrenar el perceptron con los datos de entrenamiento en caso de que no converja.
  Se pasara por parámetro en la constructora de la clase
- **weights** se trata de un diccionario que contendrá la matriz de pesos del perceptron. Este es el dato que se pretende entrenar
- **C** es una constante que delimitara el valor de tao con la finalidad de suavizar los cambios durante la actualización de la matriz de pesos consiguiendo que no sean tan bruscos. Esta preestablecida como la milésima parte

```
def trainAndTune(self, trainingData, trainingLabels, validationData, validationLabels, Cgrid):

    self.features = trainingData[0].keys()

    for iteration in range(self.max_iterations):
        print "Starting iteration ", iteration, "..."

        for i in range(len(trainingData)):
            MaxEstimacion = None
            MaxLabel = None

            for label in self.legalLabels:

                estimacion = trainingData[i] * self.weights[label]

                if ( MaxEstimacion is None or estimacion > MaxEstimacion ):
                    MaxEstimacion = estimacion
                    Maxlabel = label

            claseReal = trainingLabels[i]
            claseEstimada = Maxlabel
            if( claseReal != claseEstimada  ):

                TaoMax = self.C
                TaoAct = ( self.weights[claseEstimada] - self.weights[claseReal] ) * trainingData[i]
                                                        +
                                1.0 / 2.0*(trainingData[i]*trainingData[i])
                tao = min( TaoMax , TaoAct )

                data = trainingData[i].copy()
                for key , value in data.items():
                    data[key] = value*tao

                self.weights[claseReal]     = self.weights[claseReal] + data
                self.weights[claseEstimada] = self.weights[claseEstimada] - data
```

La función **trainAndTune** realiza el entrenamiento del perceptron

- Requerirá que se le pasen por parámetro los siguientes datos:
  - **trainingData** y **validationData** se trata de dos diccionarios que contendrán respectivamente:
    - Una matriz con todos los datos que se pretende utilizar para llevar cabo el entrenamiento del modelo.
    - Una matriz con los datos prototipo que utilizamos como modelo para las predicciones

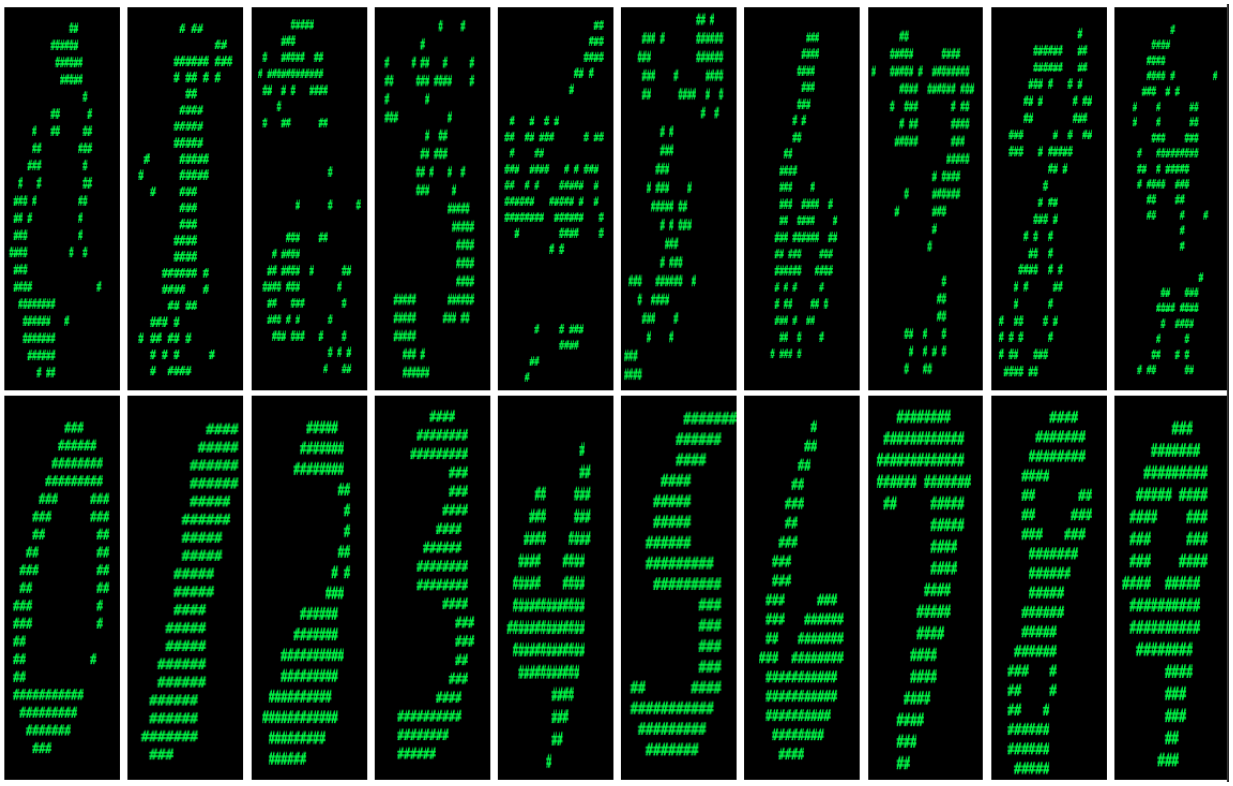
    La clave del diccionario son las coordenadas del pixel y el valor es un booleano que indica si está pintado o no

    ```
    (13, 16): 0, (2, 23): 0, (26, 8): 0, (0, 11): 0, (3, 6): 0, (27, 5): 0,
    ```
  - **trainingLabels** y **validationLabels** se trata de dos listas que contienen respectivamente:
    - La clase real de cada uno de los datos de entrenamiento
    - La clase real de los datos prototipo

    ```
    [5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9, 4, 0, 9, 1, 1,
    ```
- Se utilizaran las siguientes variables locales
  - **iteration** representa el número de veces que hemos recorrido todos los casos de entrenamiento
  - **i** representa la posición de cada una de las instancias en la matriz de entrenamiento
  - **label** representa cada una de las clases permitidas que queremos que nuestro modelo aprenda a predecir
  - **estimación** se trata del valor estimado con el que puntúa nuestro modelo a cada una de las clases permitidas
  - **MaxEstimacion** se trata del valor estimado de mayor valor
  - **MaxLabel** se trata de la clase permitida que ha obtenido la mayor puntuación
  - **ClaseReal** Es la clase que debería haber predicho nuestro modelo.
  - **ClaseEstimada** Es la clase que ha predicho nuestro modelo.
  - **taoMax** definido en la constructora para suavizar las actualizaciones de pesos
  - **taoAct** lo calculamos mediante una ponderación entre la diferencia de pesos entre la clase real y estimada y la instancia de entrenamiento que se está analizando
  - **tao** es el tao más pequeño entre el máximo que hemos establecido y el actual que hemos calculado
- Para completar el entrenamiento realizaremos las siguientes operaciones tantas veces como iteraciones se han dispuesto. (cinco en nuestro caso)
  - Por cada dato de entrenamiento
    - Reseteraremos los valores de **MaxEstimacion** y **MaxLabel** a None
    - Por cada clase permitida
      - Calcularemos el valor estimado por nuestro modelo como el producto del peso asignado a dicha clase permitida y el valor del dato de entrenamiento que estamos analizando
      - Nos guardaremos en las variables **MaxEstimacion** y **MaxLabel** los valores máximos de cada estimación y la clase a la que le corresponde dicha estimación
    - Establecemos que la **ClaseReal** es la que viene en los parámetros de entrenamiento para la instancia actual y la clase **ClaseEstimada** es la que ha obtenido la mayor puntuación por parte de nuestro modelo
    - Si la **ClaseReal** y la **ClaseEstimada** son la misma *"virgencita, virgencita que me quede como estoy"*. En caso contrario podemos asumir que nuestro modelo se ha equivocado en la predicción y procederemos a ajustar los pesos de la siguiente manera:
      - Calculamos el valor **taoAct** y comprobamos si es mayor que **taoMax.** Nos quedamos con el menor de ambos y actualizamos el valor del dato de entrenamiento que estamos analizando multiplicándolo por tao.
        - Cabe destacar que **trainingData**[**i**] es un diccionario por lo que no admite operaciones directas. De este modo recorremos sus valores uno por uno actualizándolos
      - Actualizamos los pesos de la **ClaseReal** sumándole los valores del dato de entrenamiento que estamos analizando con la finalidad de favorecer en un futuro que se prediga esta clase si se obtiene un dato como de entrenamiento como el actual.
      - Actualizamos los pesos de la **ClaseEstimada** restándole los valores del dato de entrenamiento que estamos analizando con la finalidad de penalizar en un futuro que se prediga esta clase si se obtiene un dato como de entrenamiento como el actual.

Ejecuciones

¿Cuál de las siguientes secuencias de pesos representa mejor lo aprendido por el perceptron?



python dataClassifier.py – c perceptron – w

```
(py27) C:\Users\david\Desktop\classification>python dataClassifier.py -c perceptron -w
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 ...
Validating...
56 correct out of 100 (56.0%).
Testing...
54 correct out of 100 (54.0%).
```

Conclusiones

Uno de los problemas del perceptron es que es muy sensible a cuantas iteraciones se realizan sobre los ejemplos de entrenamiento y al orden en el que se presentan los ejemplos de entrenamiento (lo mejor es que sea aleatorio)

Probaremos distintas ejecuciones para ver el porcentaje de aciertos en función del número de iteraciones tanto para el perceptron simple como para el mira perceptron

Simple Perceptron

```
Starting iteration  0 ...
85 correct out of 100 (85.0%).
*** PASS: test_cases\q1\grade.test
***     85.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
72 correct out of 100 (72.0%).
*** PASS: test_cases\q1\grade.test
***     72.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
75 correct out of 100 (75.0%).
*** PASS: test_cases\q1\grade.test
***     75.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
79 correct out of 100 (79.0%).
*** PASS: test_cases\q1\grade.test
***     79.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
76 correct out of 100 (76.0%).
*** PASS: test_cases\q1\grade.test
***     76.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
73 correct out of 100 (73.0%).
*** PASS: test_cases\q1\grade.test
***     73.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
Starting iteration  6 ...
83 correct out of 100 (83.0%).
*** PASS: test_cases\q1\grade.test
***     83.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
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 ...
83 correct out of 100 (83.0%).
*** PASS: test_cases\q1\grade.test
***     83.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
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 ...
Starting iteration  9 ...
Starting iteration  10 ...
80 correct out of 100 (80.0%).
*** PASS: test_cases\q1\grade.test
***     80.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
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 ...
Starting iteration  9 ...
Starting iteration  10 ...
Starting iteration  11 ...
Starting iteration  12 ...
Starting iteration  13 ...
Starting iteration  14 ...
Starting iteration  15 ...
83 correct out of 100 (83.0%).
*** PASS: test_cases\q1\grade.test
***     83.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

```
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 ...
Starting iteration  9 ...
Starting iteration  10 ...
Starting iteration  11 ...
Starting iteration  12 ...
Starting iteration  13 ...
Starting iteration  14 ...
Starting iteration  15 ...
Starting iteration  16 ...
Starting iteration  17 ...
Starting iteration  18 ...
Starting iteration  19 ...
Starting iteration  20 ...
84 correct out of 100 (84.0%).
*** PASS: test_cases\q1\grade.test
***     84.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points
```

Mira Perceptron

```
Starting iteration  0 ...
70 correct out of 100 (70.0%).
*** FAIL: test_cases\q3\grade.test
***     70.0 correct (0 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
64 correct out of 100 (64.0%).
*** FAIL: test_cases\q3\grade.test
***     64.0 correct (0 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
69 correct out of 100 (69.0%).
*** FAIL: test_cases\q3\grade.test
***     69.0 correct (0 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
72 correct out of 100 (72.0%).
*** FAIL: test_cases\q3\grade.test
***     72.0 correct (0 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
87 correct out of 100 (87.0%).
*** PASS: test_cases\q3\grade.test
***     87.0 correct (6 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
78 correct out of 100 (78.0%).
*** FAIL: test_cases\q3\grade.test
***     78.0 correct (0 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
Starting iteration  6 ...
81 correct out of 100 (81.0%).
*** PASS: test_cases\q3\grade.test
***     81.0 correct (6 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
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 ...
72 correct out of 100 (72.0%).
*** FAIL: test_cases\q3\grade.test
***     72.0 correct (0 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

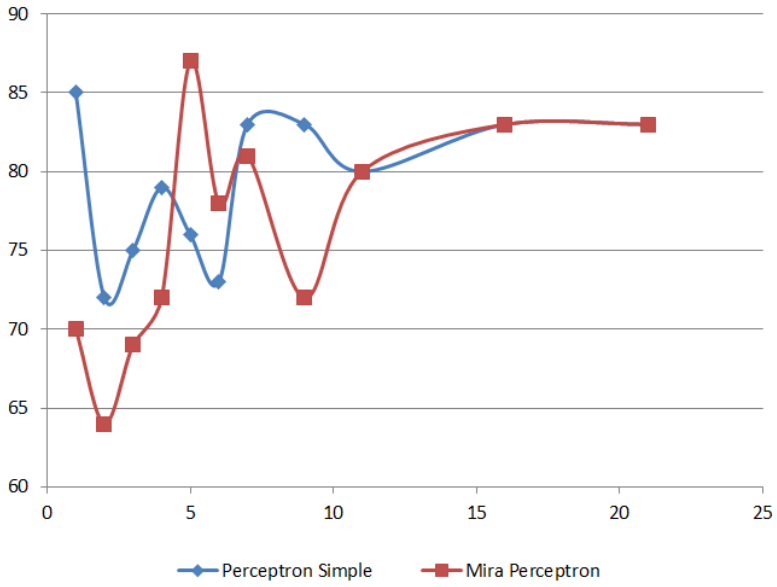
```
Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
Starting iteration  5 ...
Starting iteration  6 ...
Starting iteration  7 ...
Starting iteration  8 ...
Starting iteration  9 ...
Starting iteration  10 ...
80 correct out of 100 (80.0%).
*** PASS: test_cases\q3\grade.test
***     80.0 correct (6 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
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 ...
Starting iteration  9 ...
Starting iteration  10 ...
Starting iteration  11 ...
Starting iteration  12 ...
Starting iteration  13 ...
Starting iteration  14 ...
Starting iteration  15 ...
83 correct out of 100 (83.0%).
*** PASS: test_cases\q3\grade.test
***     83.0 correct (6 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

```
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 ...
Starting iteration  9 ...
Starting iteration  10 ...
Starting iteration  11 ...
Starting iteration  12 ...
Starting iteration  13 ...
Starting iteration  14 ...
Starting iteration  15 ...
Starting iteration  16 ...
Starting iteration  17 ...
Starting iteration  18 ...
Starting iteration  19 ...
Starting iteration  20 ...
83 correct out of 100 (83.0%).
*** PASS: test_cases\q3\grade.test
***     83.0 correct (6 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points
```

Analizando los resultados llegamos a las siguientes conclusiones:

Iteraciones	Perceptron Simple	Mira Perceptron
1	85	70
2	72	64
3	75	69
4	79	72
5	76	87
6	73	78
7	83	81
9	83	72
11	80	80
16	83	83
21	83	83



Para un número grande de iteraciones ambos métodos confluyen.

Para un número reducido de iteraciones ambos métodos funcionan bastante bien, pero el perceptron simple lo hace mejor

A medida que aumenta el número de iteraciones la eficiencia se reduce pero vuelve a aumentar nuevamente

Las representaciones siguen claramente una distribución armónica que tiende a confluir a una asíntota en el 83%

Clonación conductual

Consiste en aprender a copiar un comportamiento simplemente observando ejemplos de ese comportamiento.

En este proyecto, utilizara esta idea para imitar a varios agentes Pacman utilizando juegos grabados como ejemplos de entrenamiento.

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

    def classify(self, data ):
        guesses = []
        for datum, legalMoves in data:
            vectors = util.Counter()
            for l in legalMoves:
                vectors[l] = self.weights * datum[l]
            guesses.append(vectors.argMax())
        return guesses
```

La clase **PerceptronClassifierPacman** dispone de los siguientes atributos:

- **weights** se trata de un diccionario que contendrá la matriz de pesos del perceptron. Este es el dato que se pretende entrenar

```
def train( self, trainingData, trainingLabels, validationData, validationLabels ):

    self.features = trainingData[0][0]['Stop'].keys()

    for iteration in range(self.max_iterations):

        for i in range( len(trainingData) ):
            estado = trainingData[i]
            accionTomada = trainingLabels[i]

            movimientos = estado[0]
            accionesDisponibles = estado[1]

            for accion in accionesDisponibles:
                atributo , valor = movimientos[accion].items()[0]

                if accion == accionTomada:
                    self.weights[atributo] = self.weights[atributo] + valor
                else:
                    self.weights[atributo] = self.weights[atributo] - valor
```

La función **train** realiza el entrenamiento del perceptron

- Requerirá que se le pasen por parámetro los siguientes datos:
  - **trainingData** y **validationData** se trata de dos diccionarios que contendrán respectivamente:
    - Una matriz con todos los datos que se pretende utilizar para llevar cabo el entrenamiento del modelo.
    - Una matriz con los datos prototipo que utilizamos como modelo para las predicciones

    La clave del diccionario son las coordenadas de cada movimiento y el valor es el número de comidas pendientes
  - **trainingLabels** y **validationLabels** se trata de dos listas que contienen respectivamente:
    - La clase real de cada uno de los datos de entrenamiento
    - La clase real de los datos prototipo
- Se utilizaran las siguientes variables locales
  - **iteration** representa el número de veces que hemos recorrido todos los casos de entrenamiento
  - **i** representa la posición de cada una de las instancias en la matriz de entrenamiento
  - **estado** es la jugada que se está analizando en el instante actual del entrenamiento
  - **accionTomada** es la decisión que tomo el jugador real para solucionar la jugada actual. Por lo que es la clase real
  - **movimientos** son los posibles movimientos que habían disponibles durante la jugada actual expresado mediante un diccionario con las coordenadas de cada movimiento como clave y el número de comidas pendientes como valor
  - **accionesDisponibles** son los posibles movimientos que habían disponibles durante la jugada actual expresado mediante una lista de coordenadas
  - **atributo** solo hay uno y es el número de comidas pendientes
  - **valor** es el valor que toma el único atributo que hay

Este es un ejemplo de los datos que se utilizan

```
estado:  ({'West': {'foodCount': 27}, 'East': {'foodCount': 26}, 'Stop': {'foodCount': 27}}, ['West', 'Stop', 'East'])
accionTomada:  East
movimientos:  {'West': {'foodCount': 27}, 'East': {'foodCount': 26}, 'Stop': {'foodCount': 27}}
accionesDisponibles:  ['West', 'Stop', 'East']
   atributo:  foodCount
   valor:  27
   atributo:  foodCount
   valor:  27
   atributo:  foodCount
   valor:  26
```

- Para completar el entrenamiento realizaremos las siguientes operaciones tantas veces como iteraciones se han dispuesto:
    - Por cada dato de entrenamiento
        - Guardamos en variables los datos relativos a
            - La **accionTomada** que será la clase real
            - Los **movimientos** que son todas las posibles ejecuciones y sus consecuencias
            - Las **accionesDisponibles** que son todas las posibles direcciones que puede tomar el Pacman
        - Por cada clase una de las acciones disponibles
            - Si es la misma acción que tomo el jugador actualizamos los pesos del atributo correspondiente sumándole el valor que tomaba dicho atributo con la finalidad de favorecer en un futuro que se prediga esta clase si se obtiene un dato como de entrenamiento como el actual
            - Si no es la misma acción que tomo el jugador actualizamos los pesos del atributo correspondiente restándole el valor que tomaba dicho atributo con la finalidad de penalizar en un futuro que se prediga esta clase si se obtiene un dato como de entrenamiento como el actual.

-

Apéndices:

Resultados del Autograder

Resultados

```
Finished at 9:45:52

Provisional grades
==================
Question q1: 4/4
Question q2: 1/1
Question q3: 6/6
Question q4: 0/6
Question q5: 4/4
Question q6: 0/4
------------------
Total: 15/25

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

Apartado Q1

```
Question q1
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
76 correct out of 100 (76.0%).
*** PASS: test_cases\q1\grade.test (4 of 4 points)
***     76.0 correct (4 of 4 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  4 points

### Question q1: 4/4 ###
```

Apartado Q2

```
Question q2
===========

*** PASS: test_cases\q2\grade.test

### Question q2: 1/1 ###
```

Apartado Q3

```
Question q3
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
87 correct out of 100 (87.0%).
*** PASS: test_cases\q3\grade.test (6 of 6 points)
***     87.0 correct (6 of 6 points)
***         Grading scheme:
***          < 80:  0 points
***         >= 80:  6 points

### Question q3: 6/6 ###
```

Apartado Q4

```
Question q4
===========

*** Method not implemented: enhancedFeatureExtractorDigit at line 81 of dataClassifier.py
*** FAIL: Terminated with a string exception.

### Question q4: 0/6 ###
```

/*TODO*/

Apartado Q5

```
Question q5
===========

Starting iteration  0 ...
Starting iteration  1 ...
Starting iteration  2 ...
Starting iteration  3 ...
Starting iteration  4 ...
84 correct out of 100 (84.0%).
*** PASS: test_cases\q5\contest.test (2 of 2 points)
***     84.0 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\q5\suicide.test (2 of 2 points)
***     72.0 correct (2 of 2 points)
***         Grading scheme:
***          < 70:  0 points
***         >= 70:  2 points

### Question q5: 4/4 ###
```

Apartado Q6

```
Question q6
===========

*** Method not implemented: enhancedPacmanFeatures at line 127 of dataClassifier.py
*** FAIL: Terminated with a string exception.

### Question q6: 0/4 ###
```

/*TODO*/

Referencias

Todo el contenido teórico así como las imágenes utilizadas para este guion de prácticas han sido obtenidos de las siguientes fuentes:

Apuntes de la asignatura de IA (Ekaitz Jauregi, Eneko Agirre, Juanma Pikatza)

https://inst.eecs.berkeley.edu/~cs188/sp19

Guion de prácticas de los laboratorios