ORPerceptron Technical Document

April 7 2017

Lydia Noureldin

Algorithm

We trained a Perceptron to classify a four-input OR function, that is A v B v C v D. The Perceptron uses a few formulas during its learning process. It uses weights, wi … w4, a X input value, and a Y output value (or step value). These are calculated using the formulas below.

$$X = \sum_{i=1}^{n} w_i x_i \qquad Y = \begin{cases} +1 \text{ for } X > t \\ 0 \text{ for } X \leq t \end{cases}$$

$$Y = Step\left(\sum_{i=0}^{n} w_i x_i\right)$$

We train the weights with all combinations of four 0s and 1s (16 combinations in total). First, we generate four random weights between -1 and +1. When we analyze the first combination, the weights are adjusted until the expected value (from a normal "OR" operator) matches the Perceptron Y (output) value. If it doesn't match the expected values, we change our weights, so that newWeight = oldWeight + (learningValue * X * error). This process continues until the expected value matches our results. The weights generated by this pass are used for the second combination of the 16. After each pass, the weights are used for the next combination. After an epoch without error, the ORPerceptron is fully trained. The final trained weights are saved in a file called trained.txt and then used on the input file, in.txt. Finally, ORPerceptron.py will write the same results as the "OR" operator in the output file, out.txt.

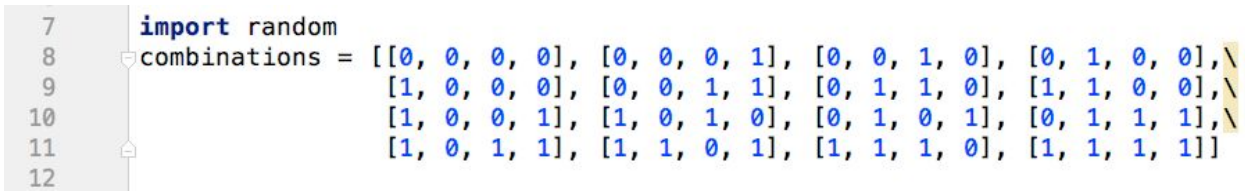
Code

Files

1. ORPerceptron_training.py
2. training.txt
3. ORPerceptron.py
4. in.txt
5. out.txt

ORPerceptron_training.py

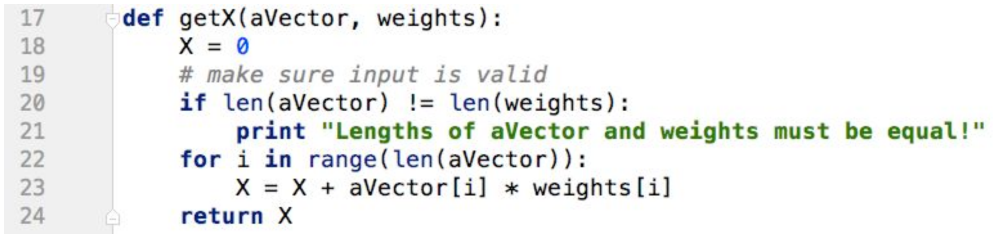
Imports and Globals



```
import random
combinations = [[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0],\
                 [1, 0, 0, 0], [0, 0, 1, 1], [0, 1, 1, 0], [1, 1, 0, 0],\
                 [1, 0, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1], [0, 1, 1, 1],\
                 [1, 0, 1, 1], [1, 1, 0, 1], [1, 1, 1, 0], [1, 1, 1, 1]]
```

random is used to generate the initial random weights between -1 and +1. Combination is a list of lists of all possible combinations of four bits used for training the Perceptron.

getX:

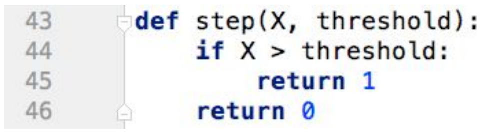


```
def getX(aVector, weights):
    X = 0
    # make sure input is valid
    if len(aVector) != len(weights):
        print "Lengths of aVector and weights must be equal!"
    for i in range(len(aVector)):
        X = X + aVector[i] * weights[i]
    return X
```

Purpose: calculates the X value given a list of weights and a vector of values.
Parameters: aVector, weights
Return: X

Uses the formula $X = \sum_{i=1}^{n} w_i x_i$ to calculate the X input value.

step:

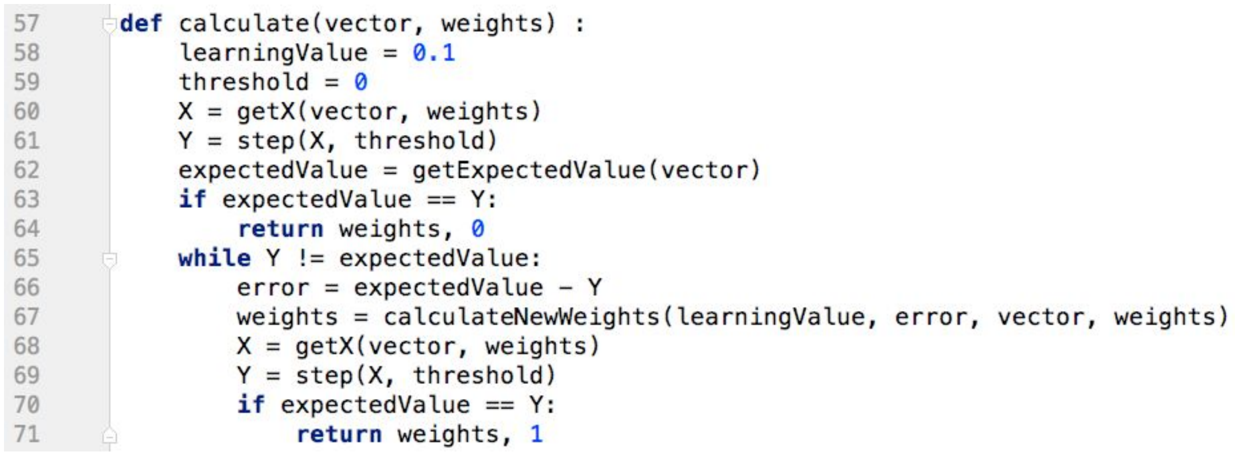


```python
def step(X, threshold):
    if X > threshold:
        return 1
    return 0
```

Purpose: Takes in X value and a threshold and calculates Y based on these values
Parameters: X, threshold (threshold is set to 0)
Return: step value

Uses the formula $Y = \begin{cases} +1 \; for \; X > t \\ 0 \; for \; X \leq t \end{cases}$ to calculate the Y output value based on the X and threshold values.

calculate:

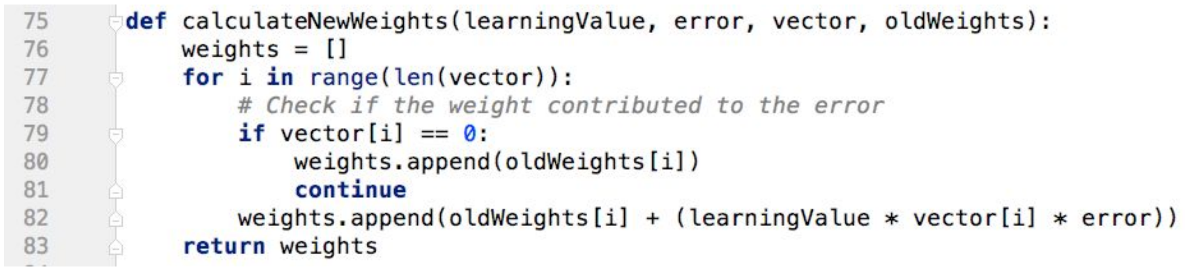


```python
def calculate(vector, weights) :
    learningValue = 0.1
    threshold = 0
    X = getX(vector, weights)
    Y = step(X, threshold)
    expectedValue = getExpectedValue(vector)
    if expectedValue == Y:
        return weights, 0
    while Y != expectedValue:
        error = expectedValue - Y
        weights = calculateNewWeights(learningValue, error, vector, weights)
        X = getX(vector, weights)
        Y = step(X, threshold)
        if expectedValue == Y:
            return weights, 1
```

Purpose: Recalculates the weights until the Y value matches the expected value
Parameters: vector, weights
Return: trained weights and, 0 if no updates were applied to the weights and 1 otherwise

First, the function calculates the X and Y values using the abovementioned functions. It then determines the expected value from an “OR” operator on the supplied vector that contains four bits. If the expected value is the same as the Y value, we have achieved our goal, and we return the weights. If they aren’t equal then we have to change the weight values. The while loop will continue to execute until Y is equivalent to the expected value. Once this is true, it returns the newly calculated weights and, whether or not they were updated.

calculateNewWeights:



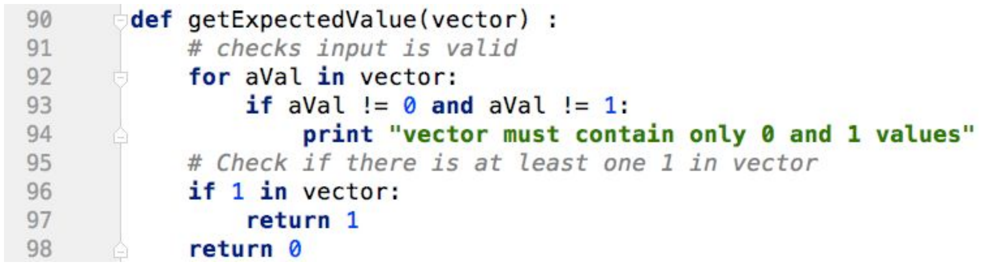
```python
def calculateNewWeights(learningValue, error, vector, oldWeights):
    weights = []
    for i in range(len(vector)):
        # Check if the weight contributed to the error
        if vector[i] == 0:
            weights.append(oldWeights[i])
            continue
        weights.append(oldWeights[i] + (learningValue * vector[i] * error))
    return weights
```

Purpose: Calculates new weights, only called if the initial randomly generated weights produced an error.
Parameters: learningValue, error, vector, oldWeights
Return: adjusted weights

Uses the formula

$w_i \leftarrow w_i + (a \times x_i \times e), \textit{where } a = \textit{learning rate, and } e \textit{ is the error value.}$ to calculate the new weights. The weight is only changed if its corresponding bit in the vector is not a 0. If it is 0 then that weight did not contribute to the problem, since anything multiplied by 0 is 0, so we skip it. It returns a list including the modified weights that contributed to the problem.

getExpectedValue:

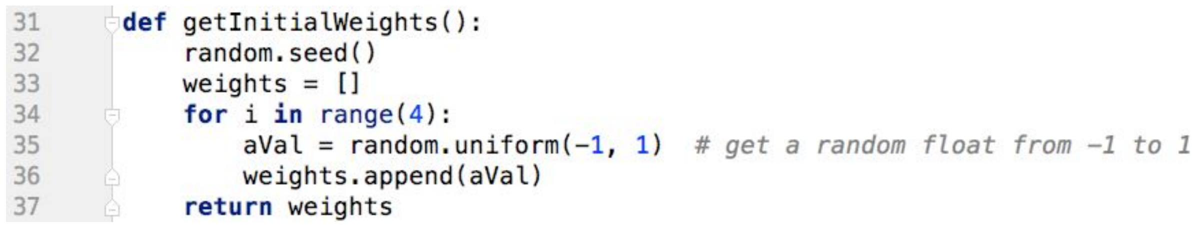


```python
def getExpectedValue(vector) :
    # checks input is valid
    for aVal in vector:
        if aVal != 0 and aVal != 1:
            print "vector must contain only 0 and 1 values"
    # Check if there is at least one 1 in vector
    if 1 in vector:
        return 1
    return 0
```

Purpose: Mimics the "OR" function, given a list of bits.
Parameters: vector
Return: the expected result based on an “OR” operator

First checks that the supplied vector only contains bit values of 0 and 1. It mimics the “OR” operator by checking if there is a 1 in the vector. If it is, then return 1. Otherwise, return 0.

getInitialWeights:

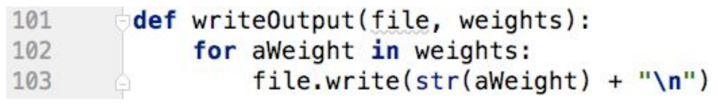


```python
def getInitialWeights():
    random.seed()
    weights = []
    for i in range(4):
        aVal = random.uniform(-1, 1)  # get a random float from -1 to 1
        weights.append(aVal)
    return weights
```

Purpose: Takes in a vector of values and returns a list of randomly generated weights of the same length of the given vector
Parameters: N/A
Return: the initial weights (before any training is done)

This is used once in the beginning of the training of the Perceptron. It generates 4 random weights between -1 and +1, one weight for each bit in the input vector. It puts all of these values in a list and returns it.

writeOutput:

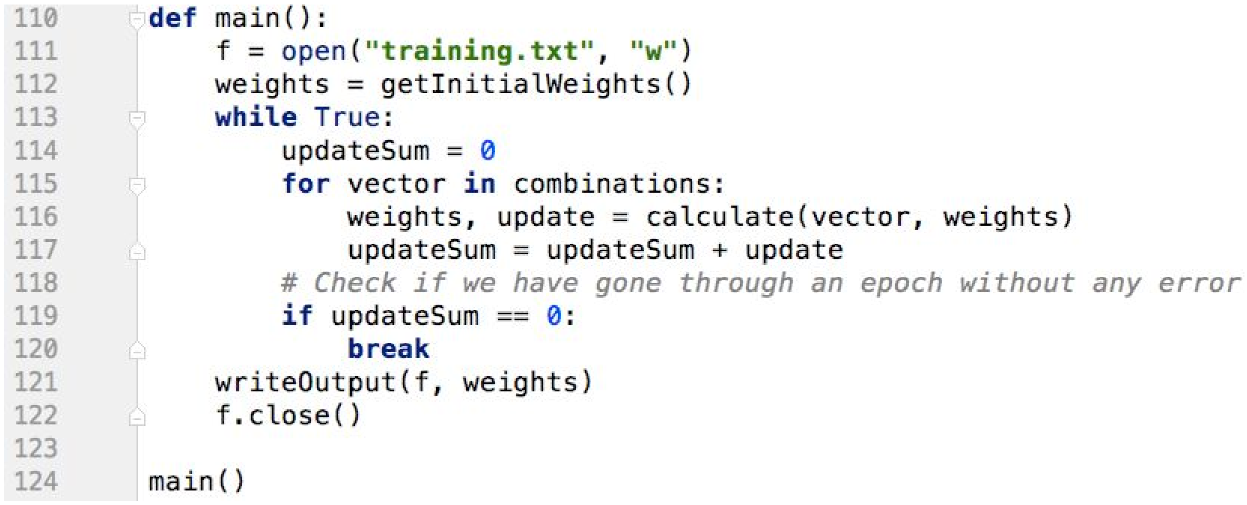


```python
def writeOutput(file, weights):
    for aWeight in weights:
        file.write(str(aWeight) + "\n")
```

Purpose: writes trained “OR” Perceptron weight values to a text file
Parameters: file, weights
Return: N/A

Takes in a file object and list of trained weights and writes them to the text file. Each weight is written on a new line.

main:



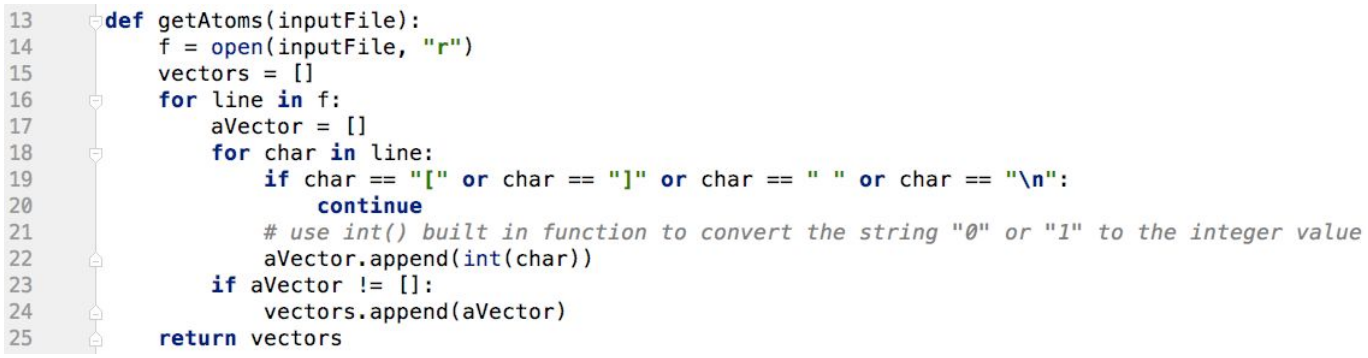
```python
def main():
    f = open("training.txt", "w")
    weights = getInitialWeights()
    while True:
        updateSum = 0
        for vector in combinations:
            weights, update = calculate(vector, weights)
            updateSum = updateSum + update
        # Check if we have gone through an epoch without any error
        if updateSum == 0:
            break
    writeOutput(f, weights)
    f.close()

main()
```

First, it opens a file called “training.txt” which will contain the trained weights after execution. It then gets the initial random weights from getInitialWeights. It will go through the 16 vectors in the combinations global constant. As it goes through the combinations, it updates the weights, and uses them on the next round. The while loop will break once the ORPerceptron is fully trained. In other words, when it goes through an epoch without any error. Lastly, it writes the trained weights to the output file and closes the file object.

ORPercptron.py

getAtoms:

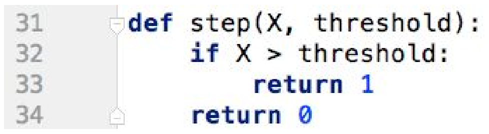


```python
def getAtoms(inputFile):
    f = open(inputFile, "r")
    vectors = []
    for line in f:
        aVector = []
        for char in line:
            if char == "[" or char == "]" or char == " " or char == "\n":
                continue
            # use int() built in function to convert the string "0" or "1" to the integer value
            aVector.append(int(char))
        if aVector != []:
            vectors.append(aVector)
    return vectors
```

Purpose: Takes in an input file of vectors separated on a new line and returns a list of lists of vectors containing the values within each vector.
Parameters: inputFile
Return: vectors

First it opens the input file for reading. Then it goes through each line and saves the numeric values within the vector in a list. For example, if the the input file contains the string “[0 0 0 0]”, then aVector will be the list of floats [0, 0, 0, 0]. This is done for each line in the input file. Each new list is saved in a list of lists called vectors. Once there are no more lines in in.txt, return vectors.

Step:

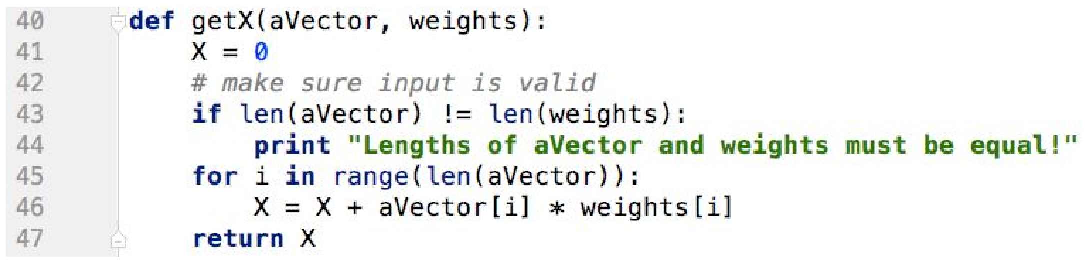


```python
def step(X, threshold):
    if X > threshold:
        return 1
    return 0
```

Purpose: Takes in X value and a threshold and calculates Y based on these values
Parameters: X, threshold (threshold is set to 0)
Return: step value

Same as ORPerceptron_training.py

getX:

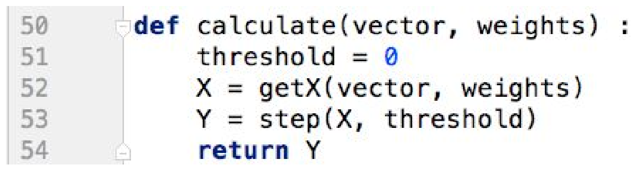


```
def getX(aVector, weights):
    X = 0
    # make sure input is valid
    if len(aVector) != len(weights):
        print "Lengths of aVector and weights must be equal!"
    for i in range(len(aVector)):
        X = X + aVector[i] * weights[i]
    return X
```

Purpose: calculates the X value given a list of weights and a vector of values.
Parameters: aVector, weights
Return: X

Same as ORPerceptron_training.py

calculate:

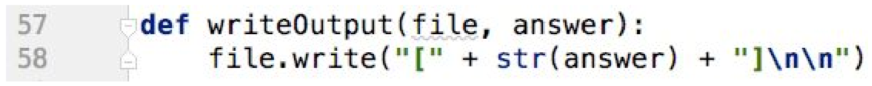


```
def calculate(vector, weights) :
    threshold = 0
    X = getX(vector, weights)
    Y = step(X, threshold)
    return Y
```

Purpose: calculates the "OR" Perceptron values based on the trained weight values
Parameters: vector, weights
Return: Y

Uses the trained weights on the vector to calculate the ORPeceptron values and returns it immediately without further weight recalculations.

writeOutput:

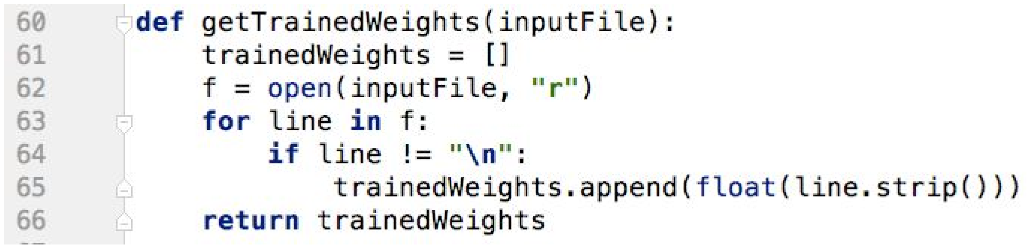


```
def writeOutput(file, answer):
    file.write("[" + str(answer) + "]\n\n")
```

Purpose: writes the "OR" Perceptron values to a text file
Parameters: file, answer
Return: N/A

Takes in a file object and answer, which is the Y value calculated using the trained weights. It writes the answer in square brackets with a blank line in between them.

getTrainedWeights:

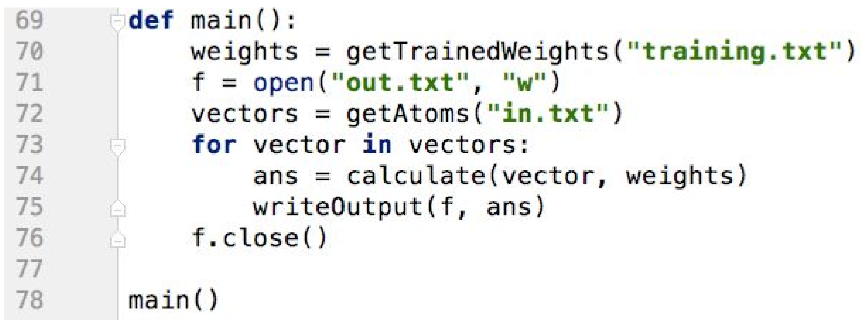


```python
def getTrainedWeights(inputFile):
    trainedWeights = []
    f = open(inputFile, "r")
    for line in f:
        if line != "\n":
            trainedWeights.append(float(line.strip()))
    return trainedWeights
```

Purpose: reads in the trained weights from an input file that are generated by ORPerceptron_training.py and saves them in a list of floats called trainedWeights.
Parameters: inputFile
Return: trainedWeights

Takes in a string of the input file's name. It opens it up for reading and saves each trained weight as a float in a list called trainedWeights. Once all the values have been read from the file, it returns trainedWeights.

main:



```python
def main():
    weights = getTrainedWeights("training.txt")
    f = open("out.txt", "w")
    vectors = getAtoms("in.txt")
    for vector in vectors:
        ans = calculate(vector, weights)
        writeOutput(f, ans)
    f.close()

main()
```

First, it gets the trained weights from the input file, training.txt. Then it opens the file out.txt to write the output of ORPeceptron. It loops through each vector from the input file and calculates the Y values and writes it to the output file. Once it has analyzed each vector, it closes the file.