Assignment 2 – Perceptron

version 1.6

This assignment is worth 30% of your course grade. You will implement the Perceptron algorithm in Python. The task is split into several functions, as specified in the section **Tasks**.

You should define all the required functions in a single Python file, named `perceptron_functions.py`. You should upload this file as individual assignment via Blackboard.

The file should only contain **function definitions** (possibly with comments), but no other executable code.

For this assignment you can use any standard Python modules, but do not use any external libraries not included with Python.

Code reuse rules

Remember that assignment 2 is individual work. You are **not allowed** to collaborate or share code with other students. **Submissions will be checked for plagiarism.**

You are welcome to study and learn from code found in online sources such as books, tutorials and blogs. You are **not allowed** to copy fragments of such code into your assignment. The only form of code reuse allowed is to call library functions.

If you are found breaking the above rules you will be reported to the Examination Board for fraud.

Grading

This assignment will be checked and graded using an automated process. Make sure you spell the function names correctly. Make sure the file you submit is accepted with no error by the Python interpreter. If your file gives errors when loaded into the interpreter, you may get zero credit. Make sure your functions give the correct answers by testing them thoroughly before submitting your assignment.

Tasks

Each of the tasks below is marked with the maximum points you can earn. Each function is also marked as *non-destructive* or *destructive*. A non-destructive function should NOT modify any of its arguments. It should build and return a new value. A destructive function modifies one or more of its arguments, for example it may change a value inside a dictionary.

Task 0 (example)

0 points

Non-destructive

Define function `words` which takes a string as an parameter and returns a list of space-delimited words in the input string. Once defined, your function should work like this:

Examples:

```
>>> words('this is an example')
['this', 'is', 'an', 'example']
>>> words('xyz')
['xyz']
```

Answer:

```
def words(input):
   "Split input string into space-separated list of words."
   return input.split(" ")
```

Task 1

1 point

Non-destructive

Define function `parse_line` which extracts the target label and feature values of a training example from a string. The string has the following format:

```
target feature_1:feature_value1 features_2:value_2 ...
```

For example:

```
1 0:2.0 3:4.0 123:1.0
```

This means the example's target label is 1, features 0 is 2.0, feature 3 is 4.0, feature 123 is 1.0 (all the other features are implicitly 0.0).

Your function should return two values:

- a dictionary mapping features (as ints) to values (as floats)
- the target label (as int)

For example:

```
>>>line = '1 0:2.0 3:4.0 123:1.0\n'
>>> print parse_line(line)
({0: 2.0, 123: 1.0, 3: 4.0}, 1)
```

Task 2

0.5 point

Non-destructive

Define function `initialize` which returns an empty perceptron model. The model should consist of a dictionary with two entries:

- *b* - the bias term of the model. The value of this key should be 0.0
- *w* - the features weights of the model. The value of this key should be an empty dictionary

(During learning, we will be updating this bias and the feature weights.)

Example:

```
>>> model = initialize()
>>> print model
{'b': 0.0, 'w': {}}
```

Task 3

1 point

Non-destructive

Define the function `dot` which calculates the dot (or inner) product of two vectors. This function should work on vectors represented as dictionaries: any missing key in the dictionary is implicitly equal to 0.0. In order to compute the dot product, you need to multiply the values at the corresponding keys together, and sum all the results. This function can assume that the vector with more non-zero entries (i.e. dictionary with more keys) will be the first argument. This is useful for efficiency.

Example:

```
>>> u = {0:0.5, 1:2.0, 2:-2.5}
>>> v = {0:-0.5, 2:2.5, 3:0.5}
>>> print dot(u, v)
-6.5
>>> w = {}
>>> print dot(u, w)
0.0
```

Task 4

1 point

Destructive

Define function `increment` which modifies a vector by adding another vector to it. The two vectors are given as dictionaries:

- *u* - the vector to be modified (as a dictionary)
- *v* - the vector (as a dictionary) which should be added to *u*

This function should not return anything, but it should modify *u* so that it contains the union of the keys present in the two vectors. The value at each key should be the sum of the values at this key in the two vectors. Remember that if a key is missing from the dictionary representing the vector, the value is implicitly equal to 0.0. Example:

```
>>> u = {0:0.5, 1:2.0, 2:-2.5}
>>> v = {0:-0.5, 2:2.5, 3:0.5}
>>> increment(u, v)
>>> print u # u has changed
{0: 0.0, 1: 2.0, 2: 0.0, 3: 0.5}
>>> u = {0:0.5, 1:2.0, 2:-2.5}
>>> w = {}
>>> increment(u, w)
{0: 0.5, 1: 2.0, 2: -2.5}
```

Task 5

1 point

Non-destructive

Define function `scale` which takes a vector *u* (as a dictionary) and a number *n*, and returns a new vector dictionary which contains the values in vector *u* multiplied by *n*. This function should not modify its

arguments, but should return a new dictionary. The function `increment` combined with the function `scale` can be used to represent vector substraction (`decrement`).

Example:

```
>>> u = {0:0.5, 1:2.0, 2:-2.5}
>>> v = {0:-0.5, 2:2.5, 3:0.5}
>>> n = 2.0
>>> print scale(u, n)
{0: 1.0, 1: 4.0, 2: -5.0}
>>> print u # u should be unchanged
{0:0.5, 1:2.0, 2:-2.5
>>> u = {0:0.5, 1:2.0, 2:-2.5}
>>> v = {0:-0.5, 2:2.5, 3:0.5}
>>> increment(u, scale(v, -1.0))
>>> print u
{0: 1.0, 1: 2.0, 2: -5.0, 3: -0.5}
```

Task 6

1 point

Non-destructive

Define function `predict` which takes two arguments:

- *model* - the dictionary representation of the perceptron model with keys 'w' for weights and 'b' for the bias
- *x* - new input (as a dictionary)

It should return the predicted target for the input *x*: it should compute the discriminant function *wx* + *b* and predict 1 if it is greater than or equal to 0, and -1 otherwise.

Task 7

2 points

Destructive

In this task we implement the update functionality of the perceptron algorithm. You need to code the function `update`, which is given a training example, and first uses the `predict` function to guess the target, and the updates the weights and the bias of the model depending on whether the guess is correct or incorrect, and on the direction of the mistake. Finally, the function should return the guess. `update` is given two arguments:

- *model* - this is the dictionary with keys 'w' (with weights) and 'b' (with the bias)
- *xy* - this is the pair *(x,y)* where *x* is the input vector (as a dictionary) and *y* is the target (-1 or 1, as an int).

Details of the perceptron update rule are shown in the lecture slides for Session 4. Hints:

a. You can use the function `predict` to make the guess.

b. When updating the weights, use the function `increment` (possibly with combination with `scale`) to add the example input to (or subtract it from) the model weights.

Example:

```
>>> x = { 0: 7.0, 1: 4.0, 3: 4.0, 4: 2.0, 5: 2.0, 11: 3.0 }
>>> y = -1
>>> model = initialize()
>>> y_pred = update(model, (x,y))
>>> print y_pred
>>> print model
1
{'b': -1.0, 'w': {0: -7.0, 1: -4.0, 3: -4.0, 4: -2.0, 5: -2.0, 11: -3.0}}
```

Task 8

1 points

Destructive

In this task you'll implement the function `learn` will process each training example, generate a guess, and make an update (using the `update` function from Task 6). Finally it will return the list of guesses made. This function is given 2 arguments:

- *model* - the dictionary representing the perceptron model
- *XY* - the list of the training examples, where each example is a tuple *(x, y)*, *x* being the input vector dictionary and *y* the target (1 or -1)

You can implement this function following these steps:

a. Initialize the list of guesses to an empty list

b. For each training example *(x,y)*

   i. get a guess using the `update` function with the *model*

   ii. add this guess to the list of guesses

c. Return the complete list of guesses.

Testing complete algorithm

Once you have correctly defined all the functions, you will be able to run the perceptron algorithm on the *sentiment* dataset. In order to do this, get the script `perceptron.py` and the datafile `sentiment.feat` from the assignment page on BlackBoard (the feature values in the datafile represent word counts in the movie reviews). Put your assignment file named `perceptron_functions.py` in the same directory as `perceptron.py` and the datafile. If you execute the `main` function in script `perceptron.py` it will run the algorithm on the data for 20 iterations and report the error rate of the predictions made within each iteration. It will also print the error rate on the development data after each iteration.

For this assigment, it is important to pay a bit of attention to speed. If your code is implemented in a very inefficient way, it will take very long to process the data. Well implemented code will complete running in under a minute.