1 Introduction

In this assignment, we will implement a perceptron learner in Python. The goal is to gain more insights about what it means to implement a learning algorithm from scratch, rather than using an existing tool like Weka. We will keep things very simple and concentrate on the conceptual aspects, rather than trying to optimize the implementation for efficiency and scalability.

Starter code: /local/kurs/ml/perc-starter-code.tar.gz

2 Linear Classification

The perceptron is a linear classifier, and we will start by implementing the basic algorithm for linear multiclass classification, that is, an algorithm that finds the highest scoring class given a set of feature weights. (Later on we will implement the algorithm to learn the weights from a training set.) The file classifier.py contains the skeleton of a linear classifier together with a simple example:

```python
def classify(x, w):
    best_class = ""
    best_score = float("-inf")
    for y in sorted(w.keys()):
        score = 0.0
        ''' Add score computation for class y here
            x = list of features
            w[y] = list of weights for class y'''
        if score > best_score:
            best_class = y
            best_score = score
    return (best_class, best_score)

if __name__ == "__main__":
    w = { "sports" : [-1.0, 0.0, 2.0],
          "news" : [0.0, 1.0, -1.0],
          "business" : [1.0, -2.0, 0.0] }
    test_data = [[1, 1, 1], [1, 1, 0], [1, 0, 0]]
    for x in test_data:
        (y, s) = classify(x, w)
        print("{} -> {} ({})".format(x, y, s))
```

The method `classify()` takes as input a list of features and a dictionary `w` containing a list of features `w[y]` for each possible class `y`. It loops over all classes, computes the score for each class, compares it to the previous best score, and in the end returns the highest-scoring class together with its score. The main program is initialized with features weights for three document classes (“sports”, “news”, “business”) and a test set of three simple instances, and uses the method `classify()` to find the best class for each instance. You can run the program as follows:

```
python3 classifier.py
```

If you run the program as is, all classes get a score of 0.0 for all instances (and the class output by the program is just the one that happens to be listed first in the dictionary). Your task is therefore to add the missing code for the scoring function, which is the core of the linear classification algorithm. Then run the program again, check that it does in fact compute real scores for the classes, and convince yourself that it finds the highest scoring class for each instance. **(Hint: There should be one document in each class.)**
3 Perceptron Learning

The weights for a linear classifier can be learned in many ways, but perceptron learning is error-driven. Every
time a training instance is misclassified, we update towards higher weights for the correct class and lower
weights for the predicted class. The file learner.py contains the skeleton of a perceptron learner together
with a simple example:

```python
from classifier import classify
from random import shuffle

def learn(data, classes, iters):
    nfeatures = len(data[0][0])
    w = {}
    for y in classes:
        w[y] = []
        for i in range(0, nfeatures):
            w[y].append(0.0)
    for n in range(0, iters):
        shuffle(data)
        for (x, y) in data:
            (z, _) = classify(x, w)
            ''' Add weight update here
            x = features
            y = correct class
            z = predicted class
            w[y] = weights for y
            w[z] = weights for z'''
    return w

if __name__ == '__main__':
    training_set = [([1, 1, 1], "sports"),
                    ([1, 1, 0], "news"),
                    ([1, 0, 0], "business")]
    test_set = [[1, 1, 1], [1, 1, 0], [1, 0, 0]]
    w = learn(training_set, ["sports", "news", "business"], 5)
    for x in test_set:
        (y, s) = classify(x, w)
        print("{} -> {} ({})".format(x, y, s))
    for y in sorted(w.keys()):
        print("{}: {}".format(y, str(w[y])))
```

First of all, note that we import the `classify()` method from classifier.py, so this will only work if
that method has been implemented correctly (and if classifier.py is in the same directory). The method
`learn()` takes three arguments: a list `data` of training instances, each of which is a pair consisting of a list
of features and a class; a list `classes` containing all the class labels; and an integer `iters` specifying the
number of training iterations. After initializing the weight dictionary `w`, it loops over the training data multiple
times, tries to classify each training instance using the current set of weights, updates the weights if necessary,
and finally returns the final weight vector. Before each training iteration, it shuffles the training set so that the
order of instances varies between iterations. The main program is initialized with a simple training set and a test
set containing exactly the same instances as the training set. It uses the training set to learn weights, uses these
weights to classify the test set, and finally prints the weights learned for each class. You can run the program
as follows:

```
python3 learner.py
```

If you run the program as is, all weights will remain set to 0.0 (and all instances in the test set will get the same
score for all classes). Your task is now to add the missing code for the weight update, which is the core of the
perceptron algorithm. Then run the program again and answer the following questions:
1. Does the perceptron learn to classify all training instances correctly?
2. If you run it multiple times, do you always get the same result? Look both at the classification and the weights learned and discuss what you see.
3. What happens if you decrease the number of training iterations? Try 1, 2, 3, 4, and 5 iterations with multiple runs in each case and discuss what you see.

4 Working with Real Data
The programs used so far are just toy examples, but the classifiers and learners you have implemented will work also for real data. The program perceptron.py (which is complete in itself) takes as input a data file, splits it into 90% for training and 10% for test, learns a perceptron classifier and reports the training and test accuracy. You can run it on (a binarized version of) the spambase data set as follows:

```python
python3 perceptron.py spambase.binary.data.txt
```
Run the experiment multiple times and report the results. Note that perceptron.py imports methods from classifier.py and learner.py, so it will only work correctly if you have implemented these methods.

5 VG Task: Averaged Perceptron
Modify learner.py to implement the averaged perceptron. Run experiments with the spambase data set and compare the results with those obtained with the simple perceptron.

6 Submission
You should submit the following:
1. Your modified versions of classifier.py and learner.py
2. A short report answering the questions in Section 3 and discussing the results of experiments in Section 4 and (optionally) Section 5