Assignment 2: Implementing a Linear Classifier from Scratch

Christian Hardmeier, VT 2019

In this assignment, we explore the workings of a linear classifier. Your starting point for the assignment is the skeleton of a Python program that reads a data set, trains a classifier, makes predictions and outputs some useful information. The code we provide takes care of many practical details that you shouldn't need to worry about, but it lacks a few essential parts to run the actual training and prediction. We ask you to implement those missing parts. You will then use the classifier in a binary sentiment classification task to predict whether movie reviews are positive or negative.

The sentiment polarity task

For this assignment, we use the Review polarity v2.0 data set created by Bo Pang and Lillian Lee at Cornell University. It consists of 2,000 movie reviews, 1,000 of which are positive and 1,000 are negative. The task is to predict, for an unseen review, whether it is positive or negative. This is a binary classification task.

As basic features, we use a bag-of-words (BOW) representation of the words in the review. Each review in the data set is described by a vector with one component corresponding to each word in the vocabulary. This component is set to 1 if the review contains this word, 0 otherwise.

Setting up your Python environment

Start by copying the Python files of this assignment to a directory in your home:

```
mkdir ml-assignment2
cd ml-assignment2
cp /local/course/ml/2019/assignment2/*.py .
```

To make sure the code runs correctly and finds the Python libraries it needs, you need to run it in a special virtual environment. Do this simply by running

```
source /local/course/ml/2019/virtualenv/bin/activate
```

from your shell prompt. You will notice that this adds the tag `(virtualenv)` to your shell prompt, indicating that your environment is set up correctly. To go back to the standard environment when you're done with the assignment, you can issue the command `deactivate`.

The Python code skeleton

The provided Python skeleton code consists of two files, `classifier.py` and `util.py`. The main parts of the classifier are in `classifier.py`. This is the only file you need to modify or even look at in order to solve this assignment. The file `util.py` contains some utility routines to load the data sets and output information. It must be present in the same directory as `classifier.py`, but you needn't worry about it otherwise. It doesn't contain anything you need to change. You can run the code using

```
python classifier.py
```

At the end of the run, the script will open a window that shows you a plot of the training progress and a histogram of the trained weights.

Start by familiarising yourself with the code in `classifier.py`. It starts with a `main` function that loads the data files and splits the data into *training*, *validation* and *test* sets, sets the training hyperparameters, calls `train` to train the classifier, checks the performance of the trained classifier on the test set and outputs some information about the results. The `predict` and `train` functions implement prediction and training. The functions `l2_norm` and `accuracy` are small helper functions that are used elsewhere in the code. Finally, the file contains four classes to implement different loss functions (`LogisticLoss` and `HingeLoss`) and regularisation methods (`L1Regulariser` and `L2Regulariser`).

It's good to know how the data is represented in the code. In this assignment, we don't use any scientific computing libraries (in the next assignment, you will). Vectors are just represented as Python lists of numbers, e.g. `[0.1, 0.3, 0.2, 1.2]`. For vector arithmetics, you would use standard Python constructs such as list comprehensions. For instance, to assign the sum of two vectors `a` and `b` to a vector `c`, you could write

```
c = [x + y for x, y in zip(a, b)]
```

This assigns to each component of `c` the sum of the corresponding components of `a` and `b`, as per the definition of vector addition.

The training, validation and test data sets are represented by objects of type `SentimentData`. These objects have a method called `features` that returns an iterator over the features of the reviews in the data set. Since almost all feature values for a given review are zero (most words in the vocabulary don't occur in most documents), we use a *sparse* representation of the features that only stores the indices of the features that are *not* zero. For each review, the iterator gives you a Python set of feature indices corresponding to the features of the review. The `SentimentData` class also has a property called `labels`. This is just a list with the correct labels for each review, 1 for positive reviews and -1 for negative reviews.

Implementing missing bits

1. **Gradient descent step**

   If you run `classifier.py` as is, you will find that it does a number of training iterations, but the numbers it outputs remain the same in every iteration. This is because it never updates the weight vector during training. Also, before starting the actual training run, the script will complain that there is something wrong with the gradient descent updates. It is your job to fix this. In the function `gradient_descent_step`, you will find a section marked `### YOUR CODE HERE ###`. There you can add the code to compute the new weight vector. Once you've done that, run the classifier again. You should now find that the gradient test report PASSED (the other tests will still fail of course) and that the training error decreases from iteration to iteration.

2. **Hinge loss**

   The code provided to you contains an implementation of the logistic loss function. It also comes with the option to select hinge loss instead, but the implementation of the loss function in the `HingeLoss` class is missing. You should implement the loss function and its gradients by filling in the two sections marked `### YOUR CODE HERE ###` in the `HingeLoss` class. Your code will automatically be tested and the program will tell you when your implementation is correct.

3. **L2 regularisation**

   The provided code contains an implementation of regularisation based on the l1 norm. The l1 norm of a vector equals the sum of the absolute values of all its components. Another very common regularisation method is l2 regularisation, using the 2-norm (or Euclidean norm) instead. The 2-norm is equal to the square root of the sum of the squared vector components. In practice, we usually drop the square root and use the squared 2-norm as our regularisation term instead. Please fill in the missing parts of the class `L2Regulariser` to enable l2 regularisation in your classifier.

Exploring hyperparameters

With the default parameters in the script, you will find that the training error improves between iterations, but the results aren't very good. Your final task is to try different parameter settings and find out how they affect the performance of your classifier. Hyperparameter changes are made by modifying the variable settings at the beginning of the `main` function. You have the following options at your disposal:

1. **Learning rate**

   Try different settings of the learning rate. It is useful to pick values from an exponentially spaced grid, e.g. 0.0001/0.0003/0.001/0.003/0.01/0.03/0.1/0.3/1.0/3.0, where each of the

values is about 3 times as large as the previous one. Note what happens when the learning rate gets too large. Try to find a learning rate that is as large as possible, but still reliable.

By default, the number of iterations is set to 5 because it doesn't make sense to run a lot of iterations before you've implemented the objective functions. Once you start experimenting with the hyperparameters, five iterations will be too few to see the full effect, so you should **increase the number of iterations**. Watch the training progress on the training and validation sets to figure out how many iterations you need for good results.

2. **Regularisation**

   You can choose between two regularisation options, **l1 regularisation** and **l2 regularisation**. Try both of them and test different values for the **regularisation strength**. An exponentially spaced grid is recommended here as well. Keep an eye on the performance of the classifier as well as the weight histogram.

3. **Loss function**

   You can choose between **logistic loss** and **hinge loss**. Try out both of them and see how they affect the results. Note that the loss values computed by the two methods are not comparable, so you should use other metrics such as accuracy to compare models between the two conditions, and that the optimal learning rate may be different depending on what type of loss you use.

Test at least 10-20 different settings in a systematic way and try to find the settings that give you the best final score (accuracy after the last iteration) on the validation set. Once you've settled on a final set of hyperparameters that work well (and only then!), set `enable_test_set_scoring` (in the hyperparameter section) to `True` and rerun to obtain a score on the test set.

VG assignment

To achieve a pass with distinction (VG) in this assignment, you must solve the tasks above without serious errors. In addition, please implement the **perceptron learning algorithm** in `classifier.py` in addition to the above and include it in your comparison. Note that the perceptron does not fit easily into the gradient descent framework, so you will have to modify the main training loop to achieve this.

Submission

Please make your submission through Studentportalen by the 25th February 2019. You should include the following:

1. Your copy of `classifier.py` with your own implementation of the missing parts.
2. A report describing the work you've done. In particular, the report should include:

- a brief description of the implementation work that was necessary to complete the missing parts, showing how you arrived at your solution
- a brief description of how and why you selected the hyperparameter values you tried
- a table listing all hyperparameter values you tested and the resulting scores (loss and accuracy) on the validation set
- the hyperparameter settings you finally selected and the resulting scores on the test set
- a discussion of what you've learnt in this assignment

Note that this assignment must be solved by each student individually.