PA4: Implementing linear classifiers

Due 5 May by 23:59 **Points** 10 **Submitting** a file upload
File types ipynb and html **Available** 25 Apr at 8:00 - 6 May at 0:15

This assignment was locked 6 May at 0:15.

To submit: a Jupyter notebook (**both in .ipynb and in .html**) containing the code of your solution as well as text cells describing what you have done. Make sure that the code is neat and the documentation readable. If you are using Colab or Deepnote, please submit the link to the project, making sure the TAs can see the content of your work.

Please **remember to join a Canvas group before submitting the assignment**. You will need to repeat this procedure for every group assignment. Also please **remember to include the names of the group members** in the notebook.

Programming assignment 4: Implementing linear classifiers

In this assignment, you will implement two algorithms to train classifiers: support vector classification and logistic regression. The pedagogical objectives of this assignment are that you should (1) get some experience of the practical considerations of implementing machine learning algorithms, (2) understand SVC and LR more thoroughly, and (3) get a taste of how a typical academic paper in machine learning looks.

In addition to the basic implementation of SVC and LR, there are a number of *bonus tasks* that let you extend the basic implementations in different ways. You need to solve at least one of these tasks to get 8p or higher (from the total maximum 10p).

Work in a group of two people. Write Python code to implement the algorithms.

You should structure your solution as a **notebook**. You have two choices: either you work on your own machine using a **Jupyter** notebook. (You can work with Jupyter directly or use the notebook support of popular IDEs such as PyCharm or VS Code.) Alternatively, you use the Colab service, which works in a similar fashion. (**Here** is a document about Colab we prepared for another course. You can ignore the GPU-related parts for now.) In case you use Colab, you can submit your solution by downloading the notebook (`.ipynb`) or by providing a link to the notebook. (In that case, make sure you allow access to the notebook.)

When you work with the notebook, you will enter your solution in the code cells. In addition, make sure that you add text cells that explain briefly what you are doing, and answer any questions you find in the assignment instructions. You should think of the notebook as a whole as a technical report.

In your solution, please make sure that the following information is included:

- The names of the people in the group.
- Your answer to the exercise questions.
- The accuracies you get for the SVC and LR classifiers, or any other classifiers you've implemented.
- Any information needed to run the code.
- Any clarification of steps in your code that could be hard to understand for someone who didn't write that code.
- Any other topic you'd like to discuss.

Preliminaries

If necessary, repeat the material from the lectures on **linear classifiers**, **gradient descent optimization**, and the **SVC** and **logistic regression** classifiers.

In this assignment, we'll implement the algorithm described in the paper **Pegasos: Primal Estimated sub-GrAdient SOlver for SVM**, Shalev-Shwartz et al. (2010). The algorithm we'll implement is described in pseudocode in Figure 1 in the paper.

In addition to the paper, you can take a look at a **separate document** that spells out the details a bit more clearly, and uses a notation more similar to what we have seen previously in the course. Make sure that you understand the pseudocode in Algorithm 1, which corresponds to Figure 1 in the original paper.

Exercise question

We are developing a simple system to predict the weather based on our location and the season. Here is a training set.

```
City:      Month:    | Weather:
-------------------------------
Gothenburg July      | rain
Gothenburg December  | rain
Paris      July      | sun
Paris      December  | rain
```

We build the system by training a perceptron from scikit-learn on this training set. We then test the classifier on the same set. As expected, the classifier correctly classifies all the instances in this set; after all, it's the same set we used to train it, so it seems that classifier has "memorized" the training set.

Now, we repeat this experiment using another training set. We train the classifier using the following examples, and again we evaluate it using the same examples we used to train it.

```
City:      Month:    | Weather:
-------------------------------
Sydney     July      | rain
Sydney     December  | sun
```

```
Paris      July      | sun
Paris      December  | rain
```

When we use this set, for some strange reason our classifier performs poorly! We can't improve it by switching to a LinearSVC. Do you have an idea what's going on? Why could the classifier "memorize" the training data in the first case, but not in the second case?

Here is the Python code if you want to try the experiment yourself.

Introduction

Download and unpack **this zip file**. The package contains the perceptron code presented during one of the lectures (`aml_perceptron.py`). Take a look at the class `Perceptron` and make sure that you understand how the Python code corresponds to the pseudocode in the lecture.

The package also contains a Python program (`doc_classification.py`) that carries out an experiment in document classification. This is the same file that we used in one of the demonstrations for one of the lectures. The task here is to determine whether a product review is positive or negative. The program trains a classifier using our perceptron implementation, and then evaluates the classifier on a test set.

Run the experiment code and make sure it works on your machine. Training should take at most a few seconds, and the accuracy should be about 0.80.

Your tasks

Implementing the SVC

Implement the Pegasos algorithm for training support vector classifiers by converting the pseudocode in Algorithm 1 in the clarification document (Figure 1 in the original Pegasos paper) into proper Python. Test your implementation by using your own classifier instead of the perceptron in `doc_clasification.py`. It's probably easiest if you start from the existing code, for instance by making a copy of the class `Perceptron`, and then just modify it to implement Algorithm 1.

To solve this task you just have to convert the pseudocode into Python code, but it can be good to try to understand why the algorithm looks as it does. Section 2 in the clarification paper goes through the steps. Understanding this may also make it a bit easier to understand what parts to change when you implement logistic regression (see below).

Find good values for the regularization parameter λ and the number of training steps (That is, either the number of iterations through the training set, or the number of randomly selected training instances if you follow the paper's pseudocode more precisely.). For instance, one might choose to iterate 10 times through the training set and set λ to $1/N$, where N is the number of instances in training set.

Sanity check: Your accuracy may vary depending on how you choose to implement the algorithm, but if the accuracy is less than 0.80 you have probably made some mistake.

Logistic regression

As we saw in the lecture, the logistic regression classifier is based on an objective function that looks almost like the one used by the SVC. The difference is just in which **loss function** is used: the SVC uses the **hinge loss** and LR the **log loss**.

Read through the explanation of the log loss and its gradient in section 3 in the clarification document. Or take a look at the table on page 15 in the Pegasos paper:

- the first line shows the hinge loss and its gradient (to be precise, the hinge loss does not have a gradient at 1. What we have here is strictly speaking a *subgradient*, not a gradient. For optimization purposes, this difference does not matter for us.), and
- the second line shows the log loss and the corresponding gradient.

Code a new training algorithm that uses the log loss instead of the hinge loss. Describe how well it works compared to your previous classifier.

Optional task: Printing the value of the objective function

Add some code to print approximate values of the SVM or LR objective function while training. For instance, if your code uses "epochs" (iterations through the training set), it seems natural to print the objective after each epoch. (Otherwise, you might print the objective value every 1,000 or 10,000 instances or so.) Recall that the objective is equal to the mean of the loss function over the whole training set, plus a regularizer term. So each time you process one training instance, you might compute the loss function as well, and add it to the sum of loss values.

Bonus tasks

To receive 8p or higher (from a total maximum 10p), it is required that you solve one of the following tasks. Please do not solve more than one task. Note that each of the bonus tasks has a number of subtasks that need to be solved.

Bonus task 1. Making your code more efficient

We kept things simple for now, but there are some ways we can make the training algorithms run faster. Try out the three approaches described below, and see if you can improve the speed. These speedups should not affect what is being computed, so you should get *the same* results with and without the speed improvements. In your report, please include time measurements so that we can see the effect of these changes. You can work with one of the SVC or LR models and it's not necessary to modify both of them.

(a) Faster linear algebra operations

The bottlenecks in the code are the linear algebra operations: computing the dot product, scaling the weight vector, and adding the feature vector to the weight vector.

Try to speed up your code by using BLAS functions, which are available in `scipy.linalg.blas`. (For general information about BLAS, see the **Wikipedia entry**.) These functions are more efficient than normal NumPy linear algebra operations, because they avoid a number of safety checks carried out

by NumPy. (The lack of these checks may cause Python to crash if you use the BLAS functions incorrectly.) The following three BLAS functions may be useful:

- If `x` and `y` are NumPy 1-dimensional arrays, then `ddot(x, y)` is equivalent to `x.dot(y)`.
- If `x` is a NumPy 1-dimensional array and `a` a number, then `dscal(a, x)` is equivalent to `x *= a`. (**Note** that this is an in-place operation, like `*=`, and will update the vector that is scaled.)
- If `x` and `y` are NumPy 1-dimensional arrays, and `a` a number, then `daxpy(x, y, a=a)` is equivalent to `y += a*x`. (This is also an in-place operation and will update `y`.)

(b) Using sparse vectors

Remove the `SelectKBest` step from the pipeline and check the difference in training time. (This will change your accuracy a bit.) You may also add the option `ngram_range=(1,2)` to the `TfidfVectorizer` and see what happens.

Our implementation is slow and wasteful in terms of memory if we use a large set of features. With sparse vectors, this can be improved. Follow the example of `SparsePerceptron` in the example code and implement sparse versions of the SVC or LR algorithms.

Hint. The helper function `sparse_dense_dot(x, w)` is used to carry out a dot product between a sparse vector `x` and a dense vector `w`, while `add_sparse_to_dense(x, w, xw)` is the counterpart of the operation `w += xw * x`.

Clarification. You don't have to use the BLAS functions you used in (a) here.

(c) Speeding up the scaling operation

At some places in the pseudocode, the whole weight vector is scaled by some factor. This can be a slow operation if our training set consists of high-dimensional, sparse vectors. Read section 4 in the clarification document, or section 2.4 in the original Pegasos paper, about a trick for speeding up the scaling operation. Modify your implementation so that it uses this trick and check if you can improve the training time.

This part can be an extension of either (a) or (b). It is probably more meaningful to extend (b) since the effect of this trick will be most noticeable for high-dimensional feature vectors.

Bonus task 2. Multiclass classification

We will see how we can make your classifiers work in a situation where we have more than two possible output classes.

Change the `read_data` function so that it uses the *first* column instead of the second column in the review file. The first column represents the type of product that is reviewed. There are six categories here: books, cameras, DVDs, health products, music and software.

(a) Binarizing the multiclass problem

Scikit-learn contains two utility classes that can help you convert multiclass classification tasks into a set of binary tasks: `OneVsRestClassifier` and `OneVsOneClassifier`. Make sure that your previous classifiers can be used with these utilities. If your classifier works correctly, you should get an accuracy of about 0.90 or a bit more.

(b) Natively multiclass learning algorithms

Consider the last two rows in the table on page 15 in the **Pegasos paper**.

Row four corresponds to a multiclass SVM. This loss function is called the multiclass hinge loss and was introduced by Crammer and Singer. (See the **Wikipedia article on the hinge loss**, under *Extensions*.)

Row five corresponds to multiclass logistic regression. The loss function is equivalent to the cross-entropy loss that we covered in **the lecture** (slide 38), just written in a more clumsy way than in the lecture.

Your task: implement multiclass SVM and LR. As usual, the gradients you need are in the right column in the table.

Hints:

- Instead of a single weight vector *w*, you probably need a matrix, or several different vectors.
- δ is a misclassification penalty. Here, we can say that it is 0 if the labels are identical, and 1 otherwise.
- In the LR model, you will need to compute the ***softmax***. You may use `scipy.special.softmax`; otherwise, make sure you don't get a numerical overflow.
- How we interpret the notation $\phi(x, y)$ depends on how you store your weights. If you have a weight matrix, then $\phi(x, y)$ means a feature vector *x* at the part of the matrix corresponding to the class *y*.

Bonus task 3. Additional loss functions

(This bonus task probably requires that you have taken a calculus course at some point.)

Derive the gradients of the loss functions described below, and plug them into the Pegasos algorithm. What is the result?

(a) Variants of the hinge loss

Look at **Wikipedia's article about the hinge loss**, and read about the *smoothed* hinge loss functions (in the section *Optimization*). They discuss a couple of different variants, but note that the first (Rennie & Srebro) is a special case of the second if you set γ to 1. These loss functions do not have the "kink" at 1 that the normal hinge loss has, which means that their gradients are continuous everywhere.

(b) Probit regression

The **probit model**, introduced by Bliss in 1934, is a binary classifier that produces a probabilistic output. However, instead of using the logistic function as in logistic regression, the probit model applies the formula

$$P(y = 1|x) = \Phi(w \cdot x)$$

where Φ is the cumulative distribution function of the normal distribution. Use the negative log likelihood as a loss function.

Hints:

- This cdf can be computed using `scipy.stats.norm.cdf` or `scipy.special.erf` (after some rescaling; see the documentation of `erf`). The latter option is probably slightly more efficient computationally.
- We have the same antisymmetric property of this model as we had for logistic regression: P(y=-1|x) = Φ(-w·x).