DATA12001 Advanced Course in Machine Learning

Exercise 3 **Due April 23, 23:59**

In this exercise set we focus on non-linear classifiers (lecture slides 6–7).

Submit pen&paper (as JPG or PDF) and application exercises (as PDF) to Moodle. Submit TMC exercises to TMC.

---

1 Kernelized SVM

TMC, 5p

Complete the KernelSVM algorithm (TMC exercise: `part3/01.svm`).

The file `svm_kernel.py` contains a partial implementation of the SVM solver described during the lectures. Provide the missing code to the functions `step` and `score`.

A non-kernel version of the algorithm is given in `svm_linear.py`. You should copy-paste the implementations of `step` and `score` given in `svm_linear.py` to `svm_kernel.py` and modify them to use kernels. You should not modify other functions.

The code in `svm_linear.py` implements (a simplified) SMO algorithm for linear SVM with slack variables (see slides). More specifically, the algorithm maintains α_n for each data point in the training data and a bias b. The score for a vector $\mathbf{x}$ is then

$$\sum \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b,$$

and the prediction for $\mathbf{x}$ is the sign of the score.

In addition the algorithm maintains $u_n = \sum \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_n$ for each datapoint $\mathbf{x}_n$ in the training data. This speeds up the computation.

The algorithm works as a coordinate ascent. It selects a pair of datapoints in training data $\mathbf{x}_i$ and $\mathbf{x}_j$, and updates α_i and α_j such that

$$\sum_n \alpha_n - \frac{1}{2} \sum_{n,m} \alpha_n \alpha_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m$$

is maximized while $0 \leq \alpha_i, \alpha_j \leq C$ and $\sum_n \alpha_n y_n = 0$ are maintained. This is done by replacing α_j with $\alpha_j + \delta$ and α_i with $\alpha_i - y_i y_j \delta$. We can then regroup the terms in the optimization criterion as

$$-\frac{1}{2}\eta\delta^2 - y_j(u_i - y_i - u_j + y_j)\delta + \text{const.},$$

where

$$\eta = \mathbf{x}_i^T \mathbf{x}_i + \mathbf{x}_j^T \mathbf{x}_j - 2\mathbf{x}_i^T \mathbf{x}_j \quad .$$

The maximum of this polynomial is $\delta^* = y_j(u_i - y_i - u_j + y_j)/\eta$. Consequently, we either use δ^* directly, or if $0 \leq \alpha_i, \alpha_j \leq C$ is violated, we select δ to be at the border of the constraint.

This maximization is already provided in `step` of `svm_linear.py`. After α_i α_j are updated, the algorithm updates u.

Once the ascent is finished, b is updated so that KKT conditions are fullfilled. This is done by finding α_i such that $0 < \alpha_i < C$. Once such α_i is found, b is set so that the score for $\mathbf{x}_i$ is equal to y_i.

To compute the kernel, use `self.kernel` which takes two data points as the input.

Hints: This is a complex algorithm but the difference between the non-kernel version and the kernel version is relatively small. Remember that kernel is used to compute the inner product between two feature vectors.

Submit your code to the TMC system.

2 Random forests

TMC, 5p

Complete the RandomizedForests algorithm (TMC exercise: `part3/02.rf`).

The file `rf.py` contains a partial implementation of random forests. To complete the implementation you need to modify the fucntions `fit` and `rf`.

The function `fit` doesn't sample features, instead it selects the best possible split. Change the code so that `fit` samples the features (without replacement), and select the best features among the sampled ones. Do this by sampling the `features` array using `np.random.choice`.

Provide the missing implementation in `rf`. The code should build `treecnt` trees, for each tree the code should sample the training data (with replacement) and construct a randomized tree using `fit`. Sample the data using `np.random.choice`. The code then should predict the test data using `predict` and compare it against the given test data. When predicting, if the total vote is equal to 0, then the sample should be counted as misclassified. To achieve this easily, compute the vote of the trees and use `np.sign` to produce the label. The function `np.sign` will return 1 if the vote is positive, -1 if the vote is negative, and 0 if the vote is 0.

The output of `rf` is as follows:

- an array `p`, where `p[i]` is the vote for the $i+1$th *test* point.
- an array `misclass`, where `misclass[i]` is the misclassication rate for the *test* data after $(i+1)$th iteration. The rate should be between 0 and 1.

Hints: There are numerous ways to implement the sampling correctly, but the automated tests specifically rely that you use `np.random.choice`. You only need one call of `np.random.choice` per sampling. You should first implement `fit` and only then `rf`, there is an automated test that tests only fit.

Submit your code to the TMC system.

3 Adaboost

TMC, 5p

Implement Adaboost algorithm (TMC exercise: `part3/03.adaboost`).

Provide the missing implementation in `adaboost` function in `src/adaboost.py`. The function takes $\mathbf{X}$ and $\mathbf{y}$ as the training data consisting of n data points, as well as the number of classifiers `itercnt`.

For the pseudo-code of the algorthm see the lecture slides.

The function should return

- an array `output`, where `output[i]` is the final weighted total vote for the $(i+1)$th data point.
- an array `err_individual`, where `err_individual[i]` is the weighted misclassification error of the $(i+1)$th *individual* classifier. The error must be between 0 and 1. In the pseudo-code `err_individual` is equivalent to ϵ.
- an array `err_ensemble`, where `err_ensemble[i]` is the misclassification error of the *ensemble* after $(i+1)$ iterations. The error must be between 0 and 1.
- an array `err_exponential`, where `err_exponential[i]` is the (normalized) exponential loss of the ensemble after $i+1$ iterations, that is, `err_exponential[i]` is equal to

  $$\frac{1}{n}\sum e^{-y_j p_j},$$

  where p_j is the weighted vote of the current ensemble for the jth data point.

As a base classifier use the provided `LDA` classifier. To train the classifier use

```
classifier = LDA()
classifier.fit(X, y, w)
pred = classifier.predict(X)
```

Submit your code to the TMC system.

4 Circle data

application, 2p

Apply the above Adaboost implementation to the provided data `toy.txt`. Construct three plots with

1. weighted misclassification error of individual classifier,
2. ensemble misclassification error,
3. normalized exponential loss

as a function of the iteration, up to 100 iterations.

Construct 4 additional plots, where you scatter plot the data and predictions, for 8, 20, 50, and 100 iterations. Use colors and markers to indicate the correct and the predicted labels.

Explain shortly what you see in all 7 plots.

Include the figures and the code, and submit your answer to moodle.

5 Neural networks

TMC, 5p

Complete the Backpropagation algorithm (TMC exercise: `part3/04.neural`).

The file `nn.py` contains a (naive) partial implementation of backpropagation algorithm.

For this exercise the network uses only sigmoid activation functions $g(x) = \frac{1}{1+e^{-x}}$ for the neurons that are not in input layer. As a loss we use $\mathcal{L}(y, o) = |y - o|$, where the label y is either 0 or 1.

The framework consists of 3 classes:

- `Edge`, representing an edge between two neurons and storing the weight of the edge.
- `Neuron`, representing a neuron, storing the bias, incoming and outgoing edges, as well as cached outputs, and gradients.
- `NN`, class representing the actual neural network.

Implement the following missing parts:

(a) `Neuron.act(self, x)`, computes the activation of `x`. Here, the activation function should be sigmoid.

(b) `Neuron.der(self, x)`, computes the derivative of the sigmoid function at `x`.

(c) `Neuron.compute_output(self)`, sets `self.input` and `self.output`. Here, `self.input` is the weighted sum of the incoming neurons, plus bias, and `self.output` and is the activated value of `self.input`. Use `self.inedges` to access the incoming neurons. You can assume that the output caches of the incoming neurons are set. Note that `self.input` may have non-zero value when `compute_output` is called, so make sure to reset `self.input` before computing the weighted sum.

(d) `Neuron.compute_delta(self)`, sets `self.delta`, equivalent to δ needed for the gradient (see the slides for more details). Use `self.outedges` to access the outgoing neurons. You can assume that the δ caches of the outgoing neurons are set. Note that `self.delta` may have non-zero value when `compute_delta` is called, so make sure to reset `self.delta` before the computation.

(e) `NN.forward(self, x)`, Forward pass: sets the input to the input layer (code already provided) and iteratively computes the outputs for all the neurons using `compute_output`. The neurons are stored in a list `self.neurons` and they are sorted, lower layers first.

(f) `NN.backward(self, y)`, Backward pass: computes δ for each neuron based on the target variable `y`. The computation of δ is already provided for the output neuron. For the remaining neurons use `compute_delta`. The neurons are stored in a list `self.neurons` and they are sorted, lower layers first.

(g) `NN.gradients(self, X, y)` computes the gradients for the weights `wgrad` and the biases `bgrad` given the training data `X`, `y`. The gradients of an individual training data point are computed by first calling `forward` and `backward` (alredy given) and then using deltas to compute the gradients. The final gradients are the sums of the gradients of individual training data points, normalized by the number of data points in the training data.

Hints: You should implement the functions in the described order. The tests will test the functions independently, that is, you can know whether `Neuron.act` is correct without implementing the other functions.

Submit your code to the TMC system.