Introduction to Machine Learning (WS2020/21)

Programming Assignment 2

Availability date: 16.11.2020
Due data: 18.12.2020

Number of tasks: 2
Maximum achievable points: 100

Learning Objectives

The learning objective of this second exercise is to to understand the process of building a "full" machine learning pipeline, including data loading and model selection, as well as advanced analysis of a classifier's performance. Furthermore, the understanding of back-propagation and foundations of gradient based (deep) learning frameworks is deepened by building an MLP regressor with custom activation function.

Submission

You have to submit both a written report and your (executable) source code. You make a submission by committing your report and source code to your gitlab repository which was created for you as part of this course (details are available in the video of the Tutorial from October 9th, 2020). Do **not** use folders or create archives of files, e.g., zip or rar. Use the following naming scheme for your files:

- The report must be a single pdf document, consisting of no more than 10 pages of size A4, using a font size of 11 pt, and 2 cm margins. Please keep the report concise—we don't expect any lengthy answers. Name your pdf report `ex2_report.pdf`.

- For each task (not each subtasks) submit a single python file name `ex2_task?.py`, where `?` is to be replaced by the task number. The python file must be executable and produce the results (plots, numbers, etc.) you use in your report.

Your code is considered as executable if it can be executed using Anaconda 3-2020.07 using only the following libraries: `numpy, scipy, sklearn, matplotlib, h5py`.

You can update any made submissions as often as you want. For grading, we will consider your latest submission made before or at 18.12.2020, 23:59:59.

Questions

If you have any questions, please ask them in the respective Moodle forum. If you don't get an answer from your colleagues or us within reasonable time (48 hours), reach out to us via email:

- Lorenz Kummer, `lorenz.kummer@univie.ac.at`
- Kevin Sidak, `kevin.sidak@univie.ac.at`
- Sebastian Tschiatschek, `sebastian.tschiatschek@univie.ac.at`

🟩 Tasks

In all tasks you are required to write Python scripts to solve the tasks. You can use all functions available in scikit-learn to solve the tasks unless stated otherwise.

Task: 1: Movie Recommendation

(maximum achievable points: 50)

In this task you will work with the "MovieLens 1M" data set [1], a relatively small benchmark dataset for recommender systems. Your goal is to train a machine learning model for recommending movies to users based on user and movie features.

Subtask 1: Data Loading and Data Preparation

(20 points)

Download the dataset from the link provided in the bibliography and unzip it. Read the included `README` file. Write functions to load and process the user information, the movie information and the ratings from `users.dat`, `movies.dat` and `ratings.dat`, respectively, and construct training data (feature vectors from user and movie data, as well as targets from the ratings) such that based on this data we can train a classifier to predict whether a user likes a particular movie or not. Consider the following points:

- The ratings are given as 1-star to 5-star ratings. Convert these ratings to binary labels, such that 4 and 5 stars ratings are mapped to label "1" and 1, 2, and 3 stars ratings are mapped to label "0". You can think of a movie-user-pair with label "1" as "the user liked the movie", while a label "0" means "the user didn't like the movie".

- When loading the data for further processing, think ahead (read the description of the next subtask first) as you will need to construct feature vectors from the data which will then be processed by various machine learning models. In particular, the binarized rating corresponds to a label we want to predict and the features you construct from the user and movie data should contain useful information for predicting that label.

- Remove all users which have rated less than 200 movies.

- Split the data into training and test data. Construct the test data from all the ratings of users with user ids $1, \ldots, 1000$ (unless a user was removed). Use the remaining ratings to construct the training data.

In the report, describe the feature vectors you created. Explain how you represented features and why? How many samples does your training and test data contain?

Hint 1: You can ignore the zip-code (in the user information) and the time stamp (in the ratings information) for simplicity.

Hint 2: Store immediate results, e.g., after loading the user file and bringing it into a suitable format, to minimize the processing time.

Remark: The specific way in which we are using the data here is not the most common one. Hence, when reading literature on recommender systems don't be confused.

Subtask 2: Basic Movie Recommendation

(15 points)

In this task you train different classifiers for predicting the binary labels constructed in the previous subtask from concatenated user-movie information, select good hyper-parameters for them, and evaluate their performance. In particular, consider linear support vector machine classifiers (`sklearn.svm.LinearSVC`) and multi-layer perceptron classifier (`sklearn.neural_network.MLPClassifier`). For the SVM tune the regularization parameter `C`, for the MLP classifiers the `hidden_layer_sizes` (number of hidden layers and their respective sizes). When tuning the hyper-parameters, use cross-validation and only use the training data. As a reference, for the MLP classifier, you should achieve a test accuracy of at least 57.5%.

Report which models you tried (e.g., state the considered hyper-parameter ranges) and which models gave you the best prediction accuracy. State the best performing models' performances on the test set. What is the best performance that can be achieved using a constant prediction of "1" or "0"?

Subtask 3: Classifier Evaluation

(15 points)

Evaluate the best classifiers from the previous task in more detail (the best SVM and the best MLP classifier). In particular, implement a function to compute and plot precision-recall-curves for the classifiers. To do so, compute classifications of the linear SVM and the MLP classifiers at different thresholds:

- For the linear SVM, you can obtain those classifications by thresholding the output of the function `decision_function`.

- For the MLP classifier, you can obtain those classifications by thresholding the output of the function `predict_proba`. This function returns probabilities indicating how likely the sample belongs to either of the two classes. You can obtain classifications for different thresholds by thresholding the probability for a sample belonging to class "1".

Don't use scikit's functions for this purpose (neither for the computation of precision and recall nor for the precision-recall curve itself) but rather implement this functionality yourself. Furthermore, implement a function to compute the average precision (AP) of your classifiers (again, don't use scikit's functions). The AP is a single number that summarizes a precision recall curve by computing the area under the curve. You can approximate the AP by approximating the area under the curve of your plot. Therefore, let R_n and P_n be recall and precision at the nth threshold. Then AP can be approximate as

$$AP \approx \sum_n (R_n - R_{n-1}) P_n.$$

Add a plot to your report, showing the precision-recall curves of the SVM and the MLP classifier on top of each other, i.e., in a single plot. Report the APs. Which classifier is best according to those metrics?

Task: 2: MLP Regression with Custom Activation Function

(maximum achievable points: 50)

In this task you will implement a multi-layer perceptron (MLP) for regression with custom activation functions, the backpropagation algorithm for computing the gradient of the loss with respect to the weights of the MLP, and train the MLP using the Adam optimizer (which is a variant of SGD that uses individual learning rates for each parameter). We consider the squared error as our loss function.

Hint 1: Read the whole description of this task before starting to code. Think about how you can implement and MLP in a modular fashion such that you can easily change the number of layers and activation functions.

Hint 2: You can solve most of this exercise without implementing backpropagation, by using a finite difference approximation of the gradient (see subtask 3). Of course, you are encouraged to implement backpropagation to get all possible points though.

Subtask 1: Regression Data

(2 points)

Download the "Regression" dataset from [3]. Once stored in a local folder, you can load the data as follows:

```
import h5py
hf = h5py.File('regression.h5', 'r')
x_train = np.array(hf.get('x_train'))
y_train = np.array(hf.get('y_train'))
x_test = np.array(hf.get('x_test'))
y_test = np.array(hf.get('y_test'))
hf.close()
```

The matrices `x_train` and `x_test` have the following shapes: (number of samples) × (number of features). The matrices `y_train` and `y_test` have the following shapes: (number of samples) × (number of targets). Normalize the data using the `MinMaxScaler`. Report the dataset statistics (number of features and training and test samples, number of targets).

Subtask 2: Forward

(8 points)

Implement forward propagation for an MLP classifier with two hidden layers and relu activations. Use small hidden layers, each containing 10 neurons. The MLP must have a single output. Initialize weights randomly as discussed in class. Initialize biases as all-zero vectors.

Briefly describe your implementation. How do you store the weights and biases? How do you store computed activations?

Hint 1: Maintain lists of layers/activations and the corresponding weight matrices and biases vectors.

Hint 2: We will later replace the activations by other activations, so keep the activations modular.

Hint 3: You might already at this stage consider that you need to store certain activations for backpropagation.

Subtask 3: Finite Difference Gradient & Optimization

(15 points)

In this task you will implement the finite difference approximation of the gradient. The forward function implemented in the previous task computes the loss function $\ell(\mathbf{x}, y; \mathbf{W}) = \ell(\mathbf{x}, y; (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots))$, where $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ are the weight matrix and bias vector of the ith hidden layer of the neural network—the number of hidden layers and the shapes of the corresponding weights and biases depend on the specific architecture. For $\epsilon > 0$, the partial derivative of the loss function with respect to weight $w_{jk}^{(i)}$ can be approximated by

$$\frac{\partial \ell(\mathbf{x}, y; \mathbf{W})}{\partial w_{jk}^{(i)}} = \frac{\ell(\mathbf{x}, y; \mathbf{W}') - \ell(\mathbf{x}, y; \mathbf{W}'')}{2\epsilon},$$

where $\mathbf{W}'$ equals $\mathbf{W}$ except for $w_{jk}^{\prime(i)}$ which is set to $w_{jk}^{(i)} + \epsilon$ and $\mathbf{W}''$ equals $\mathbf{W}$ except for $w_{jk}^{\prime\prime(i)}$ which is set to $w_{jk}^{(i)} - \epsilon$ The partial derivative with respect to bias terms can be approximated similarly.

Use the implemented approximated gradient to train the MLP. Implement training using the Adam optimizer [2]. You can import the optimizer using
`from _stochastic_optimizers.py import AdamOptimizer`
when you put `_stochastic_optimizers.py` in your local folder The optimizer can be initialized using `optimizer = AdamOptimizer(weights + biases, lr, 0.9, 0.999, 1e-08)`, where `weights` and `biases` are lists containing the weights and biases of your MLP and `lr` is the learning rate, e.g., 10^{-3} (you might need to play with the learning

rate to get good results). Once you have approximated the gradient using finite differences, you can update the parameters of the MLP using `optimizer.update_params( gradients + gradients_biases)`, where `gradients` and `gradients_biases` are lists of gradients for weights and biases respectively (using the same order as the lists you provided when initializing the optimizer). Make sure that your training converges.

Report plots showing the mean squared error on the training and test data during training. Report the time for training.

Subtask 4: Gradient Computation Using Backpropagation & Optimization

(15 points)

Implement backpropagation for computing gradients. Make sure to leverage vectorization, i.e., matrix-vector computations as discussed in class. Repeat the training from the previous task but this time use the gradient from backpropagation.

Briefly describe how you implement the backpropagation (How do you store activations? How do you compute gradients for different activation functions/layers?) Report the same plots as in the previous task. Report the time for training.

Hint: One can easily make mistakes implementing backpropagation. You can check your gradients against the finite difference approximations to spot errors.

Subtask 5: A Custom Activation Function

(10 points)

Extend your neural network to include the custom activation function $\varphi(z) = z^2$. Choose one of the weight initializations we discussed in class (report which one).

Report the partial derivative of the custom activation function.

Report performance on the training and the test set when you replace either the first or the second layer's activation function by φ. Compare the MSE of the models trained in this task with that trained in task 3 / task 4. Report the same plots as in the previous task. What do you observe when you compare the plots?

References

[1] *MovieLens 1M Dataset*. URL: `https://grouplens.org/datasets/movielens/1m/`.

[2] *Optimizers (taken from sklearn, all credits belong to the respective authors)*. URL: `https://raw.githubusercontent.com/scikit-learn/scikit-learn/master/sklearn/neural_network/_stochastic_optimizers.py`.

[3] *Regression Dataset*. URL: `https://tschiatschek.net/course/IML/WS2020/exercise2/task2/regression.h5`.