IML: Programming Assignment #2

Christian Wiskott
a01505394@unet.univie.ac.at

University of Vienna — December 20, 2020

Abstract

All tasks were completed and uploaded to the Gitlab repository. The files require no input parameters and can simply be run with a standard IDE or from the command-line. The print statements produce the relevant results.

Task1.py uses the relative path "./ml-1m/", thus requires the folder ml-1m containing the *Movie Lens* files, in the same location as the source file.

Since task2.py uses the custom class *NeuralNetwork.py*, as well as the file *_stochastic_optimizers.py*, all files must be present in the same location to run correctly.

Task 1: Movie Recommendation

Subtask 1: Data Loading and Data Preparation

In order to extract the relevant features for the recommendation model, the data was prepared via the module *pandas*, but first the data was loaded from the csv-files containing information about the users, the ratings and the movies.

For treating this problem numerically it was necessary to convert the ratings to an integer-based system. The ratings were converted to binary labels, were ratings 4 and 5 were transformed to 1, meaning the user liked the movie and 1-3 was transformed to 0, meaning, the user didn't like the movie. The genders of the users were also transformed to binary labels, such that M (male) is 0 and F (female) is 1. For taking the genre of the movies into account, a dictionary containing all genres was created which transforms the genres into integers, starting from 0 being "Action" to 17 being "Western". Due to the fact that movies can have multiple genres, the transformation was simplified, such that only the main genre i.e. the first one was considered. Later evaluations will show that despite this "loss" of information, the classifier returns sufficiently accurate results.

Next, all users with less than 200 ratings and the ratings of these users were removed and subsequently the data was split into training and test sets where users with IDs between 1 and 1000 were added to the test set and the rest to the training set. Applying this split, lead to the training set containing 561388 and the test set containing 95593 samples.

After testing the individual features by removing / adding them to the feature matrix and evaluating the accuracy of the classifier, the following features were selected: [MovieID, Genres, Gender, Age], as this combination lead to the best results. The occupation of the user didn't have any significant positive effect on the accuracy, therefore it can be assumed that the correlation between the occupation and the movies users like to watch, is not significant.

Subtask 2: Basic Movie Recommendation

For tuning the hyperparameter *C* of the SVM-Classifier, the range from 0 to 10 was evaluated and quickly showed that the optimal value lies between 0 and 1. A finer grid search revealed 0.3 as the optimal value. With the optimal parameter the maximal accuracy of the SVM-Classifier was reached at 54.2 %.

For tuning the hyperparameter of the MLP-Classifier i.e. the number and the size of the hidden layers, the range from (1,1) to (50, 100) was evaluated using a coarse search. To more accurately determine the optimal value, a finer search was conducted between (10,10) and (50,50) and revealed (25,25) as the best value. During all searches a random subset of samples have been used to evaluate the classifier, to limit the execution time. With the optimal parameter the maximal accuracy of the MLP-Classifier was reached at 57.9 %.

To keep the execution time of the py-file within sensible bounds, the coarse searches have been commented out.

Subtask 3: Classifer Evaluation

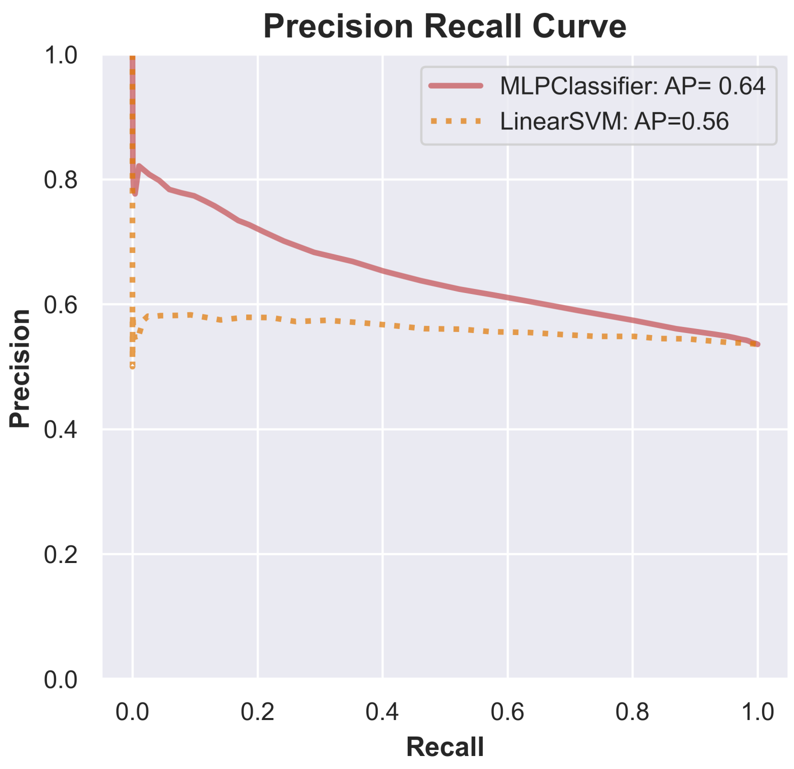

Figure 1: Precision-recall curves for the SMV and the MLP classifier. Both, the shape of the curve and the AP of the MLP reveal it as the better classifier as the SVM is not much better than a no-skill classifier.

In order to further evaluate the performance of the classifiers, the precision-recall curves as well as the APs (average precision) were plotted in fig. 1. The MLP has an AP of 0.64 and the SVM an AP of 0.56. This and the form of the PR-curve coincides with the fact that the MLP-results in higher classification accuracies than the SVM-classifier. Considering the AP value and the nearly horizontal shape of the SVM model, it suggests that it is only marginally better than a random or "no skill" classifier.

The PR-curves were obtained by calculating the precision and recall values for the different thresholds i.e. shifting the separating hyperplane and thus controlling at which point the sample is classified as class 0 or 1. The thresholds were computed using the minimum and the maximum of the confidence values of the classifiers to reach all samples with the shifting hyperplane. With this the recall and precision values were calculated for the respective thresholds:

Calculation of the thresholds

```
# Confidence values for both classifiers
dec_pred_svm = svm.best_estimator_.decision_function(xtest)
dec_pred_mlp = mlp.predict_proba(xtest)[:,1] # class 1

# Define probability thresholds to use
thresholds_svm = np.linspace(max(dec_pred_svm),
                             min(dec_pred_svm), 50)
thresholds_mlp = np.linspace(max(dec_pred_mlp),
                             min(dec_pred_mlp), 50)
```

Task 2: MLP Regression with Custom Activation Function

Subtask 1: Regression Data

The dataset "Regression", after being loaded via the *h5py* module and normalized via the *MinMaxScalar*, consists of 500 training- and 10k test-samples using 2 features.

Subtask 2: Forward Propagation

Notice: The MLP-Classifier was implemented using an object-oriented approach and thus the custom class *NeuralNetwork* was created with forward propagation as a class method. This class has been saved in a separate file named "NeuralNetwork.py". A new instance of the class can be created using the following framework, where n_hidden determines the number of hidden layers and $n\_neurons$ the number of neurons per hidden layer. In this implementation the hidden layers always have the same number of neurons and *xtrain* and *ytrain* require the shapes (features, samples) and (targets, samples), respectively.

Initialization of an instance of the NN-class

```
# Parameters for ANN
n_hidden = 2
n_neurons = 10

ann = NeuralNetwork(xtrain, ytrain, n_hidden, n_neurons)
```

The weights and biases of the respective layers, are implemented using numpy arrays and are stored in dictionaries called *W* and *b*. Every time a new instance is initiated, the following code is executed which populates *W* and *b* with the weights and biases, where the index *i* refers to the respective layer. The weights are randomly created from a normal distribution, as discussed in the lecture and the biases are initiated as zero vectors. The if-conditions are required due to the different shapes of the weight- and bias matrices of the respective layers.

```
Generation & storage of weights and biases
```

```
for i in range(1, self.n_hidden + 2): # iterates trough layers
    if i == 1: # first layer
        self.W["W{}".format(i)] = np.random.normal(0, 1,
                (self.n_neurons, self.xtrain.shape[0]))
        self.b["b{}".format(i)] = np.zeros((self.n_neurons, 1))

    elif i == self.n_hidden + 1: # last layer
        self.W["W{}".format(i)] = np.random.normal(0, 1,
                (self.ytrain.shape[0], self.n_neurons))
        self.b["b{}".format(i)] = np.zeros((self.ytrain.shape[0],
                                                          1))

    else: # every layer in between
        self.W["W{}".format(i)] = np.random.normal(0, 1,
                       (self.n_neurons, self.n_neurons))
        self.b["b{}".format(i)] = np.zeros((self.n_neurons,
                                                  1))
```

During forward propagation the computed activations are again stored in dictionaries and are returned as an argument by the forward propagation function for later use in back propagation. The activation functions were kept modular, such that they can easily be changed, but are set to the *Relu* function by default. In the interest of efficiency, the calculations are implemented using matrix operations i.e. in order to incorporate the biases in the calculations, the matrix W is extended by b via a row and the matrix X containing all sample vectors, is extended by a column of ones, in the course of the dot product.

The implementation of forward propagation can be found in the class file on lines *101 - 130*.

Info: This implementation uses *stochastic gradient descent (SGD)* with a user-defined batch size. This can be controlled via the key word *batch_size* which is set to 64 by default.

Subtask 3: Finite Difference Gradient & Optimization

In order to train the model via the finite difference approximation the class method *fit* must be called and given the appropriate parameters:

```
Parameters for Adam optimizer and finite difference method
```

```
ann.fit(xtest, ytest, max_iter=100, lr=0.15, optimizer="Adam",
method="Finite␣Difference", epsilon=0.001)
```

After thorough testing of the optimal parameters, the learning rate of 0.15 and the epsilon value of 0.001 have provided the best results. For using the Adam optimizer, the input had to be converted to lists containing the computed gradient approximations.

The implementation of the finite difference method for back propagation can be found on lines *189 - 249*, which uses dictionaries to save the results which are than used to update the weights and biases of the respective layers via the class method *update_parameters_adam* (lines *84 - 99*).

In order to add/subtract the epsilon values to the given weights and biases during the FD computation, the current *W* and *b* are copied via a *deepcopy*, to not overwrite the original matrices with the alteration. With the given altered matrices the losses are computed and used to calculate the approximations.

Info: The execution time of this method can be found in table 1 and the plot of the MSEs of the training and test set can be found in fig. 2, denoted as "Adam+FD+Relu".

Subtask 4: Gradient Computation Using Backpropagation & Optimization

To compute the back propagation via gradient computation the results of the forward propagation are stored in dictionaries which are then used to compute the gradients of the respective layers. This is done by iterating trough the layers in reverse order and again, collecting the gradients in dictionaries. After one whole iteration i.e. the gradients for all layers have been computed, the weights and biases of the model are updated via the class method *update_parameters*, which can be found on lines *77 - 82*. For reference of the gradient back propagation please refer to lines 132 - 180 of "NeuralNetwork.py".

Info: The execution time of this method can be found in table 1 and the plot of the MSEs of the training and test set can be found in fig. 2, denoted as "Adam+GD+Relu".

This method can be called by using the following function arguments:

Parameters for Adam optimizer and gradient method

```
ann.fit(xtest, ytest, max_iter=100, lr=0.15, optimizer="Adam",
method="Gradient")
```

Subtask 5: A Custom Activation Function

The custom activation function $\phi(z) = z^2$ with the partial derivative $\frac{\partial \phi(z)}{\partial z} = 2z$ was evaluated as well. The weights were initialized the same way as in the previous computations.

The custom activation was applied to the first layer and the Relu to the second, as this configuration gave the best results. The activation function can easily be adapted in the following way:

Parameters for Adam optimizer and gradient method using custom activation

```
ann.fit(xtest, ytest, lr=0.15, optimizer="Adam", method="Gradient"
, activation1=custom, d_activation1=d_custom)
```

Info: The execution time of this method can be found in table 1 and the plot of the MSEs of the training and test set can be found in fig. 2, denoted as "Adam+GD+Custom".

Fig. 2 compares the above mentioned methods via their respective MSEs. When also consulting table 1, the custom activation function combined with Adam and gradient computation performs the best in terms of MSEs as well as execution time. Despite the relatively high fluctuations of the MSEs during the first 40 iterations, this method converges to zero similarly fast as the finite difference method.

Although the finite difference method provides smaller MSEs than the gradient computation using Relu, its big downside is the excessive execution time due to the frequent loss computations and the potential overhang that comes with creating the deepcopies of the weights and biases.

The combination of Adam, gradient computation and Relu-activation is computationally more efficient than finite difference, however it requires more iterations to reach the equivalent MSEs.

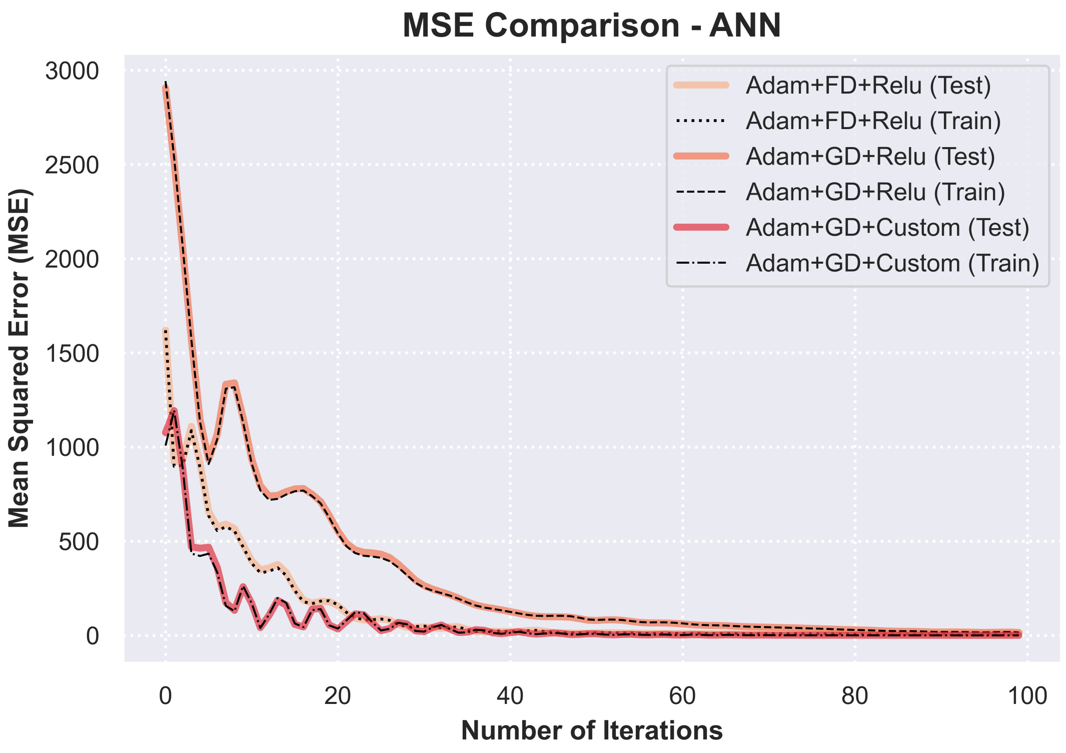

Figure 2: Comparison of the training and the test MSEs of all three considered methods. The combination of Adam optimizer, gradient computation and custom activation function applied to the first layer gives the best results. All three methods however converge to zero fairly quickly. For improved visibility of the convergence behaviour, the runs were truncated after 100 iterations. To obtain a more complete picture about the performance of the respective methods, consult the results of table 1.

Conclusion

In the course of this assignment many valuable aspects of data analysis were treated. How to prepare and manipulate data via *pandas* and ways to evaluate the performance of classifiers, as well as building a complete neural network from scratch.

Method	Test MSE	Execution Time [s]
Adam+FD+Relu	0.345	302.3
Adam+GD+Relu	0.618	39.3
Adam+GD+custom	0.002	28.2

Table 1: Performance comparison of the three chosen methods. Each evaluation was done using 10k iterations. All three methods use Adam for optimization. FD: refers to finite difference, GD: refers to gradient computation. The configuration using Adam, gradient computation and the custom function applied to the first layer provides the best results. Due to the double forward propagation per iteration and the deepcopies in the case of FD, it needs approx. 10 times longer for the same number of iterations but still manages better MESs than the gradient computation using Relu.

In task 1 it was a bit challenging to figure out the best set of features to use for classification but in the end the features [MovieID, Genres, Gender, Age] resulted in 57.9% MLP classification accuracy. It was shown that the MLP-classifier provided the best results w.r.t. the classification accuracies as well as the shape of the PR-curve and the AP value.

Task 2 proved to be much more difficult and evolved and took extensive testing to figure out the correct implementation that could use the Adam optimizer in combination with custom activation functions and finite difference. The combination of Adam, gradient computation and the custom activation on the first layer provided the best results relative to the other two methods, with the smallest test-MSE of 0.02 and the shortest execution time of 28.2s after 10k iterations.