# Lab Course Machine Learning

**Exercise Sheet 6**

Prof. Dr. Dr. Lars Schmidt-Thieme
Jung Min Choi

HiWi: Harish Malik
Submission deadline : November 28, 2024

### General Instructions

1. Data should be normalized.
2. Train to Test split should be 80-20 / with Validaiton 70-15-15
3. Convert any non-numeric values to numeric values. For example you can replace a country name with an integer value or more appropriately use one-hot encoding.

## 1 Naive Bayesian Classifier (10 points)

In this assignment, you will implement a Naive Bayesian Classifier from scratch and compare its performance with the implementation from `sklearn`.

1. **[6 Points]** Implement a Naive Bayesian Classifier from scratch. Use the provided dataset `logistic.csv`. Load the data and split it into training and test sets using `train_test_split` from `sklearn`. Feature scaling can also be applied using `sklearn` methods if you think it necessary.
2. **[2 Points]** Compute the following evaluation metrics. (do not use `sklearn` or any other library for these computations):
   - Accuracy
   - Precision
   - Recall
   - F1-Score
3. **[2 Points]** Train and evaluate the Naive Bayesian Classifier implemented using `sklearn`'s `GaussianNB`. Compare the evaluation metrics (accuracy, precision, recall, F1-score) between your custom implementation and the `sklearn` implementation.

## 2 MLP Regressor with PyTorch (10 points)

You will build and train a simple Multilayer Perceptron (MLP) regressor using the California Housing dataset.

1. **[1 Point]** Download the California Housing dataset using `sklearn`. Perform the following preprocessing steps:
   - Split the dataset into training, validation, and test sets.
   - Apply feature scaling (e.g., normalization or standardization)
   - You can use sklearn.
2. **[6 Points]** Build and train a Multilayer Perceptron (MLP) regressor using `PyTorch`. The MLP should satisfy the following requirements:
   - The architecture should allow customization of:
     - The number of hidden layers ($n$).
     - The activation function (e.g., `ReLU`, `Tanh`, etc.).

- Use the Adam optimizer and Mean Squared Error (MSE) as the loss function.
- Use validation dataset for training.
- You can use torch for the optimizer and the loss function.

3. **[1 Point]** Evaluate the model on the test dataset using MSE.
4. **[2 Point]** Plot the training, and validation loss trajectories.

**Hints:**

1. **PyTorch Tutorials and Installation:**
   - Students who are unfamiliar with PyTorch can explore the official tutorials at the following link: https://pytorch.org/tutorials/.
   - To install PyTorch, refer to the installation guide available here: https://pytorch.org/get-started/locally/.
2. **Pseudocode for MLP Regressor:**
   - Define the MLP architecture:
     - Create a class that inherits from `torch.nn.Module`.
     - Define the layers of the MLP (input, hidden, and output) using `torch.nn.Linear`.
     - Use an activation function (e.g., `torch.nn.ReLU`) after each hidden layer.
   - Implement the forward pass in the class.
   - Preprocess the data:
     - Split the dataset into training, validation, and test sets.
     - Normalize or standardize the features.
     - Convert the data to PyTorch tensors.
     - Use `torch.utils.data.DataLoader` to create batches for training and validation.
   - Define the optimizer (e.g., `torch.optim.Adam`) and loss function (e.g., `torch.nn.MSELoss`).
   - Train the model:
     - Loop over a specified number of epochs.
     - For each epoch:
       * Perform forward and backward passes for training data.
       * Compute and track losses for training and validation datasets.
   - Evaluate the model on the test set and compute MSE and RMSE.
3. **PyTorch Functions:**
   - **Data Handling:**
     - `torch.utils.data.DataLoader` for creating data batches.
     - `torch.tensor` to convert numpy arrays or data into PyTorch tensors.
   - **Model Definition:**
     - `torch.nn.Module` to define the MLP model.
     - `torch.nn.Linear` for fully connected layers.
     - `torch.nn.ModuleList` to manage layers dynamically.
     - Activation functions like `torch.nn.ReLU`, `torch.nn.Tanh`, etc.
   - **Training Components:**
     - `torch.optim.Adam` for the optimizer.
     - `torch.nn.MSELoss` for the Mean Squared Error loss function.
   - **Utilities:**
     - `torch.no_grad` to evaluate the model without computing gradients.

# 3 **Bonus: Optuna Hyperparameter Tuning (5 points)

You will implement hyperparameter tuning for the MLP Regressor using `Optuna`. The goal is to optimize key hyperparameters to achieve the best performance on the validation dataset and evaluate the tuned model on the test dataset.

1. **[2 Points]** Add a `Dropout` layer to your MLP Regressor architecture:
   - Incorporate `torch.nn.Dropout` to regularize the model and prevent overfitting.
   - The dropout rate should be one of the hyperparameters you tune.
2. **[3 Points]** Implement hyperparameter tuning using `Optuna`:
   - Optimize the following hyperparameters:
     - Learning rate.
     - Activation function (e.g., `ReLU`, `Tanh`).
     - Number of hidden layers.
     - Dropout rate.
   - Use the training and validation datasets to tune hyperparameters. Report every hyperparameter tuning trial.
   - Select the best hyperparameters based on the validation loss.
   - After finding the best hyperparameters, train the MLP Regressor using the training and validation datasets combined, and evaluate its performance on the test dataset. Report the following metrics:
     - Mean Squared Error (MSE)
     - Mean Absolute Error (MAE)