Neural and Evolutionary Learning

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

Predicting Lactose Content in Milk Using Automatic Milking System Data

Afonso Anjos, 20230527

Filipe Pereira, 20230445

João Machado, 20230426

June, 2024

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

1. Introduction
This project aims to compare and analyze the performance of various models studied in the Neural and Evolutionary Learning course. Using a dataset from a North Italian farm, which includes data on cow milk production obtained through Automatic Milking Systems, the primary objective is to predict lactose content in milk. We preprocess the dataset to handle missing values and select appropriate features. We then compare five models: Genetic Programming (GP), Geometric Semantic Genetic Programming (GSGP), Neural Networks (NN) with backpropagation, NeuroEvolution with GPOL, and NeuroEvolution of Augmenting Topologies (NEAT). The comparison involves statistical analysis to determine the most effective approach for predicting lactose content, thereby enhancing the assessment of milk quality in automated farming systems.

2. Data preprocessing
In the data preprocessing our objective was to look at the data before we started creating any models, in order to change and improve it as much as necessary to be ready for future tasks. To do this, we observed the different statistics of the numeric variables, as well as the possible missing values, where we noticed that almost half of the rows in the variable "dry_days" were missing, and so we deleted this column. We also plotted various histograms to check the distributions, as well as a correlation matrix, where we noticed the variables "lactation" and "delivery_age_years" were highly correlated, and so we chose to delete "delivery_age_years". Finally, we decided to scale the data (features) using "Robust Scaler" due to the outliers we had previously detected when plotting various boxplots.

3. Implementations and library changes
3.1. Initializers
In the initializers, we did some minor changes to the existing code. We changed the full and grow functions to full_individual and grow_individual, respectively. After this we created two new functions, full and grow. These two functions were very similar, as they were just made to create a population of solutions by repeatedly calling full_individual and grow_individual, respectively.

3.2. Selectors
For the selectors, we did some changes to the existing code by wrapping the roullete_wheel, rank_selection and rnd_selection with wrapper functions, like we had in the tournament function, so we could use them as intended. We also implemented a new function, prm_double_tournament.

3.2.1 prm_double_tournament
The prm_double_tournament function creates a double tournament selection algorithm for genetic programming or genetic algorithms, optimizing based on RMSE and tree_size. It takes two parameters: pressure1 and pressure2, which set the selection pressures for the two tournaments. It returns the double_tournament function. First, a helper function tournament1 performs selection based on the first objective (RMSE). The main double_tournament function uses tournament1 to select individuals for the first objective. From these, it randomly selects a subset based on pressure2 for the second tournament. The best individual in this subset is then selected based on the second objective (tree_size), either minimizing or maximizing as required.

3.3. Variators
For the variators, we did some changes to the existing code by wrapping the hoist_mtn function with prm_hoist_mtn so we could use it as intended, and we also implemented two new functions, nn_xo and prm_nn_mtn.

3.3.1 nn_xo

The nn_xo function performs a crossover on the weights and biases of two parent neural networks (p1 and p2) to produce two offspring. It extracts and flattens the weights and biases from each layer of both parents. A random crossover point is chosen, and segments before this point are swapped between the parents to create new arrays for the offspring. These modified arrays are reshaped back to their original dimensions and added to the offspring's weight and bias lists. Finally, the function combines these new weights and biases into tuples representing the two offspring and returns them, ensuring a mix of parameters from both parents.

3.3.2 prm_nn_mtn

The prm_nn_mtn function creates a mutation operator for neural network weights and biases. It takes a mutation scale ms and a configuration dictionary sspace. The inner function nn_mtn performs the mutation by adding random noise to the weights and biases. For each layer, it generates random tensors scaled by ms and adds them to the original parameters. This produces new, mutated weights and biases. The mutated parameters are returned as a new neural network, introducing necessary randomness and variation to improve optimization.

4. Models

As we delve into the modeling phase, a key aspect is optimizing hyperparameters, which greatly influence model performance. We begin by preparing our data for training. Then, we embark on hyperparameter tuning using a randomized search approach. We chose this approach as it would not be feasible to test all possible combinations due to the vast number of potential combinations. For each combination of the random search, we executed the model with the specified parameters and recorded the performance based on the RMSE of the model on the validation set. Throughout this process, we aim to find hyperparameter settings that maximize performance while ensuring robustness and generalization.

4.1. Genetic Programming (GP)

For this model, we've incorporated a range of parameters in our grid, covering function sets, probability of constants, and tree depths. Additionally, we've carefully considered genetic algorithm parameters like population sizes, mutation probabilities, crossover probabilities, selection pressures, elitism, and reproduction. Alongside these, we've also taken into account various initializers, selectors, crossover methods, and mutators. After conducting the random search, the optimal combination of hyperparameters was determined to be:

- Function set: (add, sub, mul, div, log); Probability of constants: 0.1; Maximum initial depth: 3; Maximum depth: 15; Initializer: rhh; Selector: prm_double_tournament; Crossover: swap_xo; Mutator: prm_hoist_mtn; Population size: 100; Selection pressure: 0.05; Mutation probability: 0.05; Crossover probability: 0.8; Elitism: True; Reproduction: True.

These hyperparameters resulted in a validation root mean square error (RMSE) of 0.0627.

4.2. Geometric Semantic Geometric Programming (GSGP)

For this model, the same grid as in GP was employed, and after conducting the random search, the optimal combination of hyperparameters was determined to be:

- Function set: (add, sub, mul, div, log); Probability of constants: 0.05; Maximum initial depth: 3; Maximum depth: 20; Initializer: full; Selector: prm_double_tournament; Crossover: prm_gs_xo; Mutator: prm_gs_mtn; Population size: 500; Selection pressure: 0.1; Mutation probability: 0.01; Crossover probability: 0.7; Elitism: True; Reproduction: True.

These hyperparameters resulted in a validation root mean square error (RMSE) of 1.2003.

4.2.1 Efficient Geometric Semantic Geometric Programming (GSGP)

For this model, the same grid as in GP was used, and after conducting the random search, the optimal combination of hyperparameters was determined to be:

- Function set: (add, sub, mul, div, exp, tanh); Probability of constants: 0.2; Maximum initial depth: 3; Maximum depth: 15; Initializer: full; Selector: prm_double_tournament; Crossover: prm_efficient_gs_xo; Mutator: prm_efficient_gs_mtn; Population size: 250; Selection pressure: 0.1; Mutation probability: 0.1; Crossover probability: 0.8; Elitism: True; Reproduction: False.

These hyperparameters resulted in a validation root mean square error (RMSE) of 0.0722.

4.3. Neural Networks with backpropagation (NN)

For this model, our initial approach was the implementation of a simple neural network (NN) architecture, iterating over 100 epochs and using backpropagation. The model's hyperparameters were optimized through a search process, testing combinations of hyperparameters using k-fold cross-validation, aimed at decreasing the core loss function of mean squared error (MSE). In addition to the simple NN architecture, we also took into account the impact of training duration by experimenting with different training epochs for the simple NN architecture The best combination found in the hyperparameter search underwent training and validation on the training data, where we tracked the performance by using root mean squared error (RMSE), in order to monitor overfitting, using plots, for both the training and validation sets. Additionally, more complex neural networks were created, having multiple hidden layers and dropout regularization in some of them. Similar optimization steps, as the first simple NN, were applied to identify the best hyperparameters for all of these different models. Our best neural network ended up being the complex NN with 200 epochs:

- Architecture: input layer with 12 input features; first hidden layer with 128 neurons; followed by ReLU activation function; followed by a Dropout layer; second hidden layer with 32 neurons, followed by another ReLU activation; final output layer for regression (1 neuron).
- Hyperparameters: Hidden Layer 1 Size: 128 neurons; Hidden Layer 2 Size: 32 neurons; Learning Rate: 0.1; Optimizer: SGD (Stochastic Gradient Descent); Dropout Probability: 0.5

This combination resulted in a validation root mean square error (RMSE) of 0.0600.

4.4. Neuroevolution with GPOL-based framework

For this model, we've incorporated a range of parameters in our grid, covering initial factors, hidden neuron configurations, and activation functions. Additionally, we've carefully considered genetic algorithm parameters like population sizes, mutation steps, mutation probabilities, crossover probabilities, selection pressures, elitism, and reproduction. Alongside these, we've also taken into account selectors, including prm_tournament, prm_roulette_wheel, prm_rank_selection, and prm_rnd_selection. After conducting the random search, the optimal combination of hyperparameters was determined to be:

- init_factor: 0.1; n_hidden_neurons: [8, 8]; activation: [torch.tanh, torch.tanh, None]; selector: prm_tournament; ps: 250; mutation_step: 0.25; mutation_prob: 0.1; xo_prob: 0.2; selection_pressure: 0.1; has_elitism: False; allow_reproduction: False.

These hyperparameters resulted in a validation root mean square error (RMSE) of 0.0672.

4.5. NeuroEvolution of Augmenting Topologies (NEAT)

In our parameter grid, we've included ranges for neural network parameters such as activation mutate rate, aggregation mutate rate, bias mutate rate, weight mutate rate, node add probability, and connection add probability. We've also considered genetic algorithm parameters including population size, elitism, and survival threshold. The best parameters identified through the random search process are as follows:

- Population Size: 500; Activation Mutation Rate: 0.5; Aggregation Mutation Rate: 0.7; Bias Mutation Rate: 0.1; Weight Mutation Rate: 0.7; Node Addition Probability: 0.2; Connection Addition Probability: 0.7; Elitism: 2; Survival Threshold: 0.1.

These parameter settings resulted in a validation root mean square error (RMSE) of 0.0518.

5. Evaluation and results

Once we determined the optimal parameters for all our models, we proceeded to evaluate their performance on the test set. This crucial step allowed us to assess how well our models generalize to unseen data and validate their effectiveness.

Model	GP	GSGP	GSGP_Efficient	NN	NE_GPOL	NEAT
RMSE	0.0646	1.3562	0.0716	0.0600	0.0724	0.0587

Table 1 - Model Performance Comparison based on RMSE

Despite the variation in performance across the models, NEAT stands out for achieving the lowest RMSE score among the algorithms evaluated.

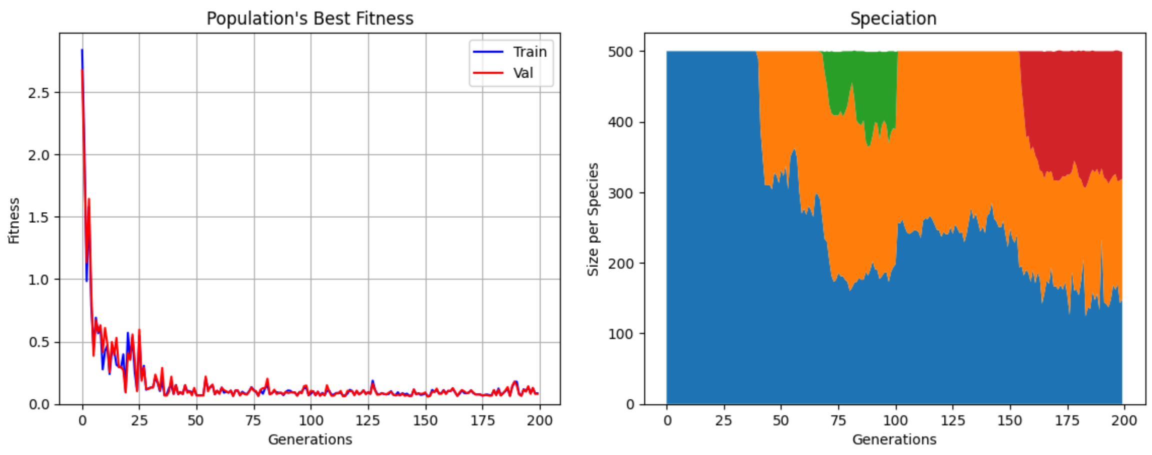

Figure 1 - Fitness and speciation evolution of NEAT

While NEAT emerged as the top performer, it's worth noting that other algorithms, such as GP, Efficient GSGP, NN, and NE_GPOL, also demonstrated competitive performance with relatively low RMSE scores. The small differences in RMSE between these models suggest that they are all viable options, depending on other considerations such as computational efficiency, interpretability, or ease of implementation. In contrast, GSGP exhibited significantly higher RMSE compared to the other algorithms, indicating poorer performance in generalizing to unseen data. This was also the model that took the most time, due to its high solution complexity. For that reason we could only run 10 generations instead of 50, which could have affected its performance.