Milk Components and Quality Inference

Automatic Milking Systems Problem

Utilizing Neuroevolutionary Algorithms to infer milk components

Authors:

André Filipe Silva	20230972
Mariana Cabral	20230532
Sebastião Rosalino	20230372
Sofia Pereira	20230568

NOVA IMS

Academic Orientation: Leonardo Vanneschi and Karina Brotto Rebuli

2023/2024

Contents

1 Introduction

Leveraging the power of algorithms to address societal challenges is becoming increasingly common in today's world. This project focuses on utilizing such models to create more efficient and automated *Automatic Milking Systems (AMS)*, based on data from a farm in Northern Italy. During each session, the milking robot collects extensive data on individual cow productivity and milking behavior. Important metrics such as fat, protein, and lactose levels are measured to evaluate milk quality. However, these measurements are currently obtained using colorimetric methods that require calibration every two weeks.

The goal of this project is to infer lactose content using data from the milking robot. If this approach can be scaled to include fat and protein, it could eliminate the need for traditional measurement methods, thereby simplifying the evaluation of milk quality.

2 Data Exploration and Preprocessing:

For this project, the following documents were provided: a dataset (*data_project_nel.csv*) composed by 324 records and 14 numerical features and 3 targets (*y_fat.csv*, *y_protein.csv* and *y_lactose.csv*). Although the main goal of the project is to infer lactose and the group focused on this target variable, throughout the follow-up stages, this step was conducted keeping in mind the three target variables.

From the initial exploration it was noted the presence of 147 missing values in the variable *dry_days*, which correspond to observations from the first lactation. This could be justified by the fact that cows do not have a dry period before their first lactation.

Additionally, upon examining histograms and boxplots of the features, a certain amount of outliers were observed, but given the purpose of this course and the reality behind the problem and data, they can be justified or correspond to infrequent observations. Furthermore, the small dataset size led us not to remove these observations and instead, test different algorithms, evaluating their impact on the models. If considered relevant to improve the models, the treatment of outliers could be conducted later.

Regarding multivariate analysis, two strategies were applied. First, correlation matrices among features to inspect redundancy and relevancy, looking for both linear (*Pearson's*) and nonlinear (*Spearman's*) relations. Next, scatter plots were generated to visualize the relationships between each feature and the targets, looking for relevance.

In the preprocessing phase, the group focused on the principal concern: the presence of missing values. Given their nature, solving this issue meant choosing a problem definition from three available options. The group opted for a more conservative strategy that retained more features and records, i.e chose to keep the *dry_days* feature. This decision required removing the records from the first lactation period, resulting in a dataset with 177 records. To explore the impact of the remaining lactation levels, dummy variables corresponding to each lactation level were created.

After the feature engineering, correlations among the resulting variables were checked, following the same approach as before. As a result of this analysis, the *dim* variable was removed for not being related to any other variable. Finally, the dataset was inspected for duplicates, not being found any.

3 Genetic Programming and Double Tournament Implementation

For **Genetic Programming**, it was implemented the standard class available in the *gpolnel* library. Since the dataset was rather small, it was utilized Cross-Validation for the algorithm, as it constitutes a more robust way to make use of all data and more confidently assess the performance of the models, compared to the more common split in train, validation and test subsets.

The model was initially configured using the *grow* function from *initializers.py* (Grow initialization). To examine the impact of using both Full and Ramped half-and-half (RHH) initializations, a *full()* function was developed, mirroring the approach of the Grow method. This function produces *n_sols* individuals via the *full_individual()* function. The RHH initialization was implemented as presented in class.

With this said, it was tested GP with the three initializers (*Grow*, *RHH*, *Full*). For the *Grow* initializer, it was tested a *max_init_depth* of 3 and 5. This parameter controls the maximum depth of trees during the initial population generation. This was done in order to test and strike a balance in solution complexity. Complex solutions are prone to overfitting, and the value of 3 produced better results - it was used as default from this point onwards. The remaining parameters were maintained as default.

It was also implemented **Double Tournament Selection**, to apply in GP and GSGP. In essence, this is a two-stage method, where, in the first stage, two tournaments with a random K number of individuals from the population are conducted simultaneously, with one winner being extracted from each tournament based on the best fitness (since this is a minimization problem, the best fitness corresponds to the smallest RMSE). Those two *winners* proceed to the second stage, where they face each other in a tournament and the winner is picked based on tree size - the one with smallest tree size is the winner.

This was implemented directly in the *gpolnel* library, specifically in the *selection.py* file. Two main things changed: an extra import was added (*from gpolnel.utils.tree import Tree*) and a function called *prm_double_tournament()* was added to the code. You can also find the code of this function in the final notebook.

4 Geometric Semantic Genetic Programming

For ***GSGP***, the *GPOL-based Efficient Semantic Genetic Programming* was implemented. This method uses Efficient Operators to address the rapid growth of individuals, proving to be the most effective solution for this problem.

In deploying *GSGP* with *Cross-Validation*, a similar approach to that used in *GP* was adopted. The same initial search space was used, with the function set expanded to include *exp* and *log*, while keeping the algorithm's parameters consistent with the previous model.

To create the Supervised Machine Learning Problem instance, the *SMLGS* class from *inductive_programming.py* was utilized. Unlike the previously discussed algorithms, *Efficient GSGP* only requires the dataset as input, bypassing the need for torch data loaders and batch training, using only the train/val indices from each fold's split.For the algorithm's instance, the *GSGP* class from *genetic_algorithms.py* was used to define the final model, maintaining all parameters used in *GP*, except for the crossover and mutation operators. These were replaced by the efficient operators *prm_efficient_gs_xo* and *prm_efficient_gs_mtn* from *variators.py*, with Grow initialization and a mutation step (ms) equal to values from the interval from 0.25 to 5, incremented 0.25, randomly selected during mutation.

To optimize results, the model was trained with a higher probability of selecting a constant as a tree terminal (0.4), a smaller max depth for initializing the trees (2), increased selection pressure (0.2), and a higher mutation probability (0.3). This strategy emphasizes greater variability and exploration through increased use of constants, larger tournament sizes, and higher mutation rates. The best model was also tested using Full and RHH initializations, and with Double Tournament applied to all initializations.

5 Neural Network

Based on the implementations given during classes, the group decided to deploy a **Neural Network** using *Pytorch*. To address this problem, the network was changed to regression type, using the *RegressionNN* class, that inherits from *nn.Module* (the base class for all neural network modules in *PyTorch*) and defines the feed-forward neural network.

Within that class, the forward function and the initialization of weights and biases were implemented similarly to the labs. However, the weight initialization was not performed inside the class's___*init*___ method, being moved to the cross-validation to facilitate the assessment of the algorithm's performance through multiple runs. This approach ensured that the weights are initialized only once per model run, avoiding re-initialization at each epoch, which would disrupt the training process by resetting the weights.

The architecture was simplified, having just an input layer, a hidden layer activated by the *ReLU* function , and an output layer. It should be mentioned that, according to the convergence of the model with this architecture, others might be interesting to test.

Additionally *train_model* and *evaluate_model* functions were implemented to help the training and evaluation of the neural network, respectively. Both functions were designed to compute the loss and Root Mean Squared Error (RMSE) for better performance assessment. During training, the *train_model* function sets the model to training mode, iterates through the data loader, and performs a forward pass, loss computation, backpropagation, and weight update using the optimizer. It also calculates the RMSE for each batch and averages it over the entire data loader. Similarly, the *evaluate_model* function sets the model to evaluation mode and computes the loss and RMSE without updating the model weights, ensuring the integrity of the evaluation process. By including RMSE in both functions, the performance of the model can be effectively monitored and compared across different epochs and validation sets.

Finally, smaller learning rates were tested, looking to achieve models with slower convergence: learning rates of 0.001, 0.00001, and 0.0000001 were each tested with 500 epochs to observe the difference in the evolution of the models, but the change showed no impact in the convergence.

6 NeuroEvolution of Augmenting Topologies (NEAT)

To enhance the predictive power of the Neural Networks model, the **NEAT** algorithm was implemented due to its ability to evolve both the topology and weights of neural networks. Given the small dataset size, the initial attempt was to use 10-fold cross-validation but faced implementation challenges. Consequently, it was adopted a **train/test split** approach, dividing the data into training (70%) and testing (30%) sets. Continuous features were scaled using a **Robust Scaler**, fitted on the training dataset to avoid data leakage. The choice of a Robust Scaler, instead of more common techniques such as Min-Max or Standard Scalers, was driven by the aim of mitigating the influence of outliers, which were not removed during the preprocessing stage, thereby enhancing the algorithm's generability to unseen data.

The NEAT was configured based on the class configuration *config-feedforward-xor* file, however with the following adaptions to make it ready to use for the **regression** problem: *'fitness_criterion'* set to *'min'*, since the group wants to minimize the absolute RMSE value; *'fitness_threshold'* set to '0.00000001', providing NEAT with virtually unlimited flexibility in searching for the best neural network;*'pop_size'* set to '200';*'activation_options'* set to *'identity'*;*'num_inputs'* set to '19', to include the one-hot encoded lactation variables;

Furthermore, it was used an evaluation function to compute the RMSE between the predicted and actual lactose content. The fitness function was set to minimize the RMSE, guiding the evolutionary process to produce networks with better predictive performance. Each genome in the population was evaluated using these functions, and the best-performing genome was selected based on its fitness score.

To ensure robustness, and to compensate for not using K-fold cross-validation, NEAT was executed 30 times with different random states. This iterative process involved splitting the data, scaling features, and evolving the network for 100 generations in each run.

7 Model Comparison and Results

To test the algorithms with statistical significance, 30 trials for each model were run, which means having 30 values of the RMSE for each algorithm (Single Tournament GP with Grow Init, RHH, Full, ..., all the way to NEAT). Given all those results the first employed test was**Friedman's test**, to identify if there were statistically significant differences among the results. The null hypothesis was rejected, as such confirming that these differences were significant.

Then, focus on knowing which, out of all the model results were different, and given those, what were the best performers, the group employed **Wilcoxon's signed-rank test**, in a 1v1 fashion among all the models. This test tells if the median of two paired groups is identically distributed or not. By applying this, and when the null hypothesis was rejected (hence, the model results were different with statistical significance), the group chose the best model out of the pair by checking the means of all the RMSEs collected and choosing the lowest mean as the winner. The winner would then proceed to face against other models, until it found one that outperformed it.

Algorithm	GP ST	GSGP ST	GP DT	GSGP DT	NN	NEAT
30 Trials Mean RMSE	0.00145	0.16688	0.10123	0.15246	0.42892	0.06017

Table 1: Mean RMSE over 30 trials for all algorithms.
Note: GP and GSGP only present the results for the Grow Initializer as that is always the best out of the 3 implemented.

Through this procedure, the best performing model was **Single Tournament Selection Genetic Programming, Grow Init, max_init_depth = 3**, with an average RMSE over 30 runs of 0.00145.

This result was surprising as the group assumed that a model like NEAT, or even the implementation of Double Tournament Selection would lead to better results. However, the '*no free lunch theorem*' holds, and for this particular problem, this was the best algorithm. The group believes that the small data and feature size played in favour of the winning model. With extra complexity, it is possible that a different algorithm would have achieved better results.

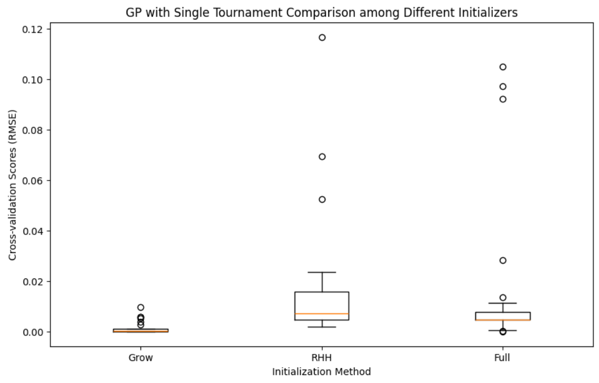

Figure 1: Results for different initializations in Single Tournament Genetic Programming

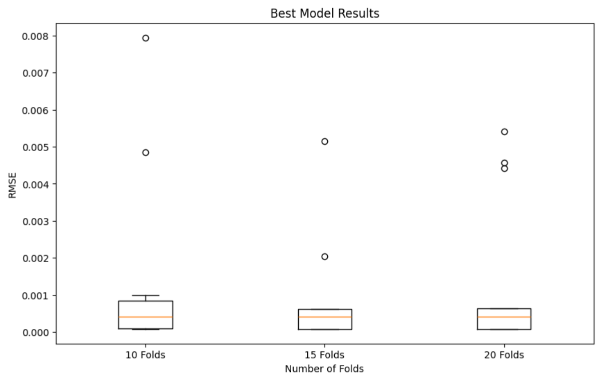

Figure 2: Best Model Results, evaluated with different number of folds

8 Conclusion

The project aimed to infer lactose content using data from *AMS*, and this objective was successfully achieved as the models developed demonstrated an *RMSE* remarkably close to zero. This high level of accuracy may be attributed to the relative simplicity of the problem, as most algorithms explored achieved good performance and converged rapidly. However, this simplicity also posed a limitation, restricting the variety of tests that could be performed for each algorithm.

Future work could explore several alternative approaches. These include incorporating different mutations and crossovers, studying and predicting protein and fat variables to enhance model performance, and testing all three problem approaches to create a comprehensive and efficient solution encompassing all lactation periods.