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

Index

1. Data Exploration

In [1]:
```
# Import the necessary libraries and configurations
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import os
import csv
from configparser import ConfigParser
from tempfile import NamedTemporaryFile
import time
import psutil
import threading
import sys
import gc
import itertools

from IPython.display import display, Javascript
from tqdm import tqdm

import sklearn
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset
```

```
from gpolnel.problems.inductive_programming import SML, SMLGS
from gpolnel.utils.ffunctions import Ffunctions
from gpolnel.utils.inductive_programming import function_map
from gpolnel.algorithms.genetic_algorithm import GeneticAlgorithm, GSGP
from gpolnel.operators.initializers import grow, prm_grow, ERC, full, prm_full, rhh
from gpolnel.operators.variators import swap_xo, prm_subtree_mtn, prm_efficient_gs_
from gpolnel.operators.selectors import prm_tournament, prm_double_tournament
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
import torch.nn.functional as F

from scipy.stats import wilcoxon, friedmanchisquare

import neat
import visualize
```

In [2]:
```
start_time = time.time()
```

In [3]:
```
# Function to save the notebook
def save_notebook():
    display(Javascript('IPython.notebook.save_checkpoint();'))

# Function to get system memory usage percentage
def system_memory_usage_pct():
    memory = psutil.virtual_memory()
    return memory.percent  # Percentage of total memory usage

# Function to check system memory and act if threshold exceeded
def check_system_memory(percent_threshold, check_interval=1):
    while True:
        mem_usage_pct = system_memory_usage_pct()
        if mem_usage_pct > percent_threshold:
            print(f"System memory threshold exceeded at {mem_usage_pct}%. Saving no
            save_notebook()
            time.sleep(5)  # Wait a couple of seconds to ensure the notebook is sav
            os._exit(0)
        time.sleep(check_interval)

# Start the memory check in a separate thread
def start_memory_monitor(percent_threshold=90, check_interval=5):
    thread = threading.Thread(target=check_system_memory, args=(percent_threshold,
    thread.daemon = True  # Set the thread as a daemon so it will close when the ma
    thread.start()
    return thread

# Usage
monitor_thread = start_memory_monitor(98)  # Check if system memory usage exceeds 9
```

In [4]:
```
# Load the datasets
train = pd.read_csv('datasets/data_project_nel.csv')
y_fat = pd.read_csv('datasets/y_fat.csv')
```

```
y_lactose = pd.read_csv('datasets/y_lactose.csv')
y_protein = pd.read_csv('datasets/y_protein.csv')
```

In [5]:
```
# Check the first rows of the train dataset
train.head()
```

Out[5]:

	lactation	delivery_age_years	dim	dry_days	forage_kg_day	rumination_min_day	milk_k
0	7	7.750000	414	56.0	4.310918	434.814010	34.0
1	8	9.083333	357	78.0	4.167087	589.500000	36.1
2	6	7.666667	315	69.0	4.903333	542.577778	43.3
3	7	8.666667	362	69.0	4.294724	628.371901	41.6
4	5	6.500000	427	58.0	4.798618	479.334112	37.9

In [6]:
```
# Check the first rows of the y_fat dataset
y_fat.head()
```

Out[6]:

	fat_percent
0	3.787156
1	3.645519
2	3.458251
3	3.407140
4	4.902554

In [7]:
```
# Check the first rows of the y_lactose dataset
y_lactose.head()
```

Out[7]:

	lactose_percent
0	4.953503
1	4.983128
2	4.889104
3	4.868969
4	4.845402

In [8]:
```
# Check the first rows of the y_protein dataset
y_protein.head()
```

Out[8]:

	protein_percent
0	3.511685
1	3.470806
2	3.370124
3	3.221164
4	3.395152

In [9]:
```
# Check structural information about the train dataset, such as feature names, non-
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 324 entries, 0 to 323
Data columns (total 14 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   lactation                 324 non-null    int64  
 1   delivery_age_years        324 non-null    float64
 2   dim                       324 non-null    int64  
 3   dry_days                  177 non-null    float64
 4   forage_kg_day             324 non-null    float64
 5   rumination_min_day        324 non-null    float64
 6   milk_kg_day               324 non-null    float64
 7   milk_kg_min_robot         324 non-null    float64
 8   milkings_day              324 non-null    float64
 9   errors_by_100_milkings    324 non-null    float64
 10  high_cdt_by_100_milkings  324 non-null    float64
 11  watery_by_100_milkings    324 non-null    float64
 12  refusals_by_milking       324 non-null    float64
 13  colostrum_separated_kg    324 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 35.6 KB
```

In [10]:
```
# Check the shape of the y_fat dataset
y_fat.shape
```

Out[10]:
```
(324, 1)
```

In [11]:
```
# Confirm that there are no missing values in the fat percentage target
y_fat.isna().sum()
```

Out[11]:
```
fat_percent    0
dtype: int64
```

In [12]:
```
# Check the shape of the y_lactose dataset
y_lactose.shape
```

Out[12]:
```
(324, 1)
```

In [13]:
```
# Confirm that there are no missing values in the lactose percentage target
y_lactose.isna().sum()
```

Out[13]:
```
lactose_percent    0
dtype: int64
```

In [14]:
```
# Check the shape of the y_protein dataset
y_protein.shape
```

Out[14]:
```
(324, 1)
```

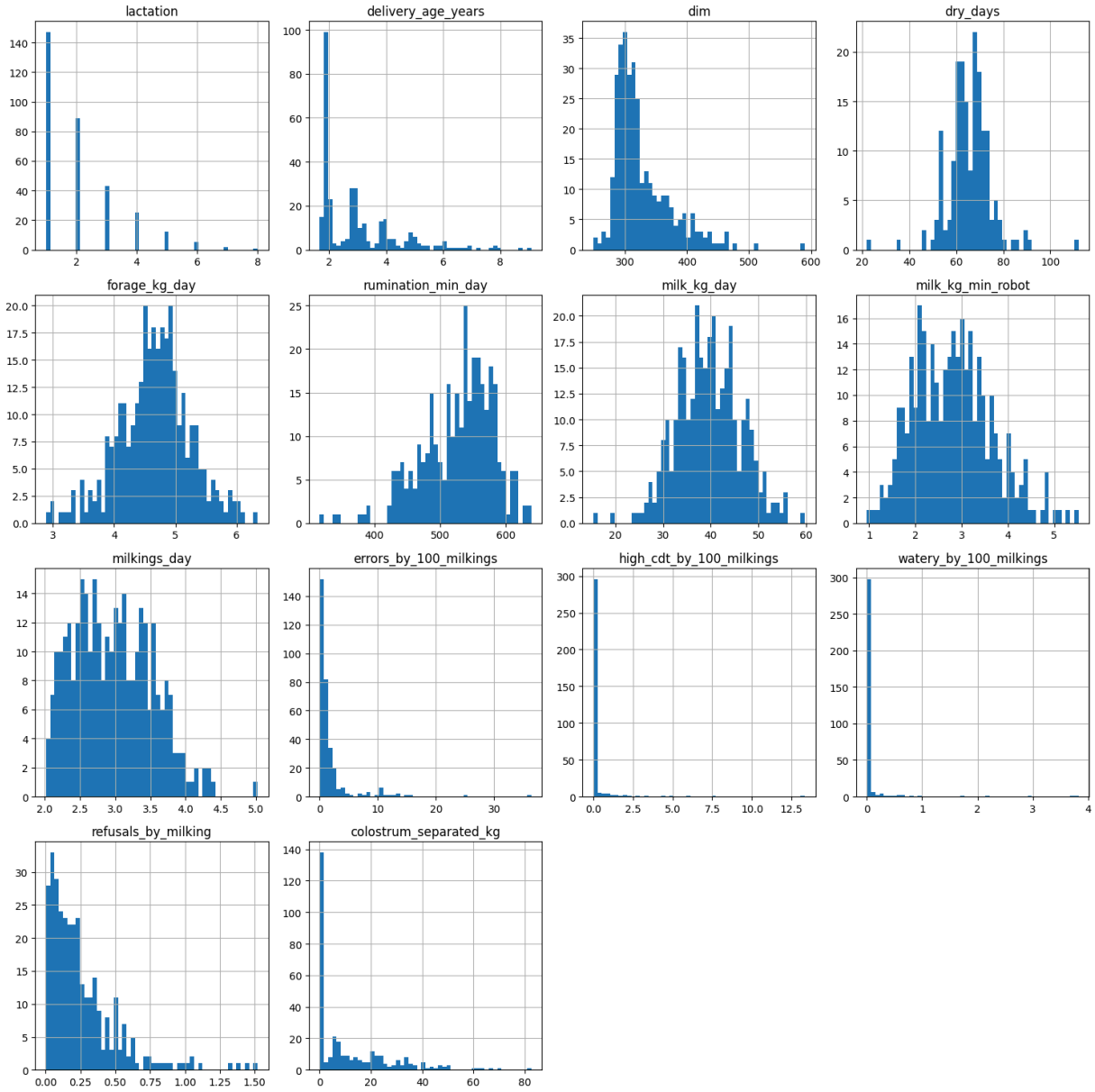
In [15]:
```
# Confirm that there are no missing values in the protein percentage target
y_protein.isna().sum()
```

Out[15]:
```
protein_percent    0
dtype: int64
```

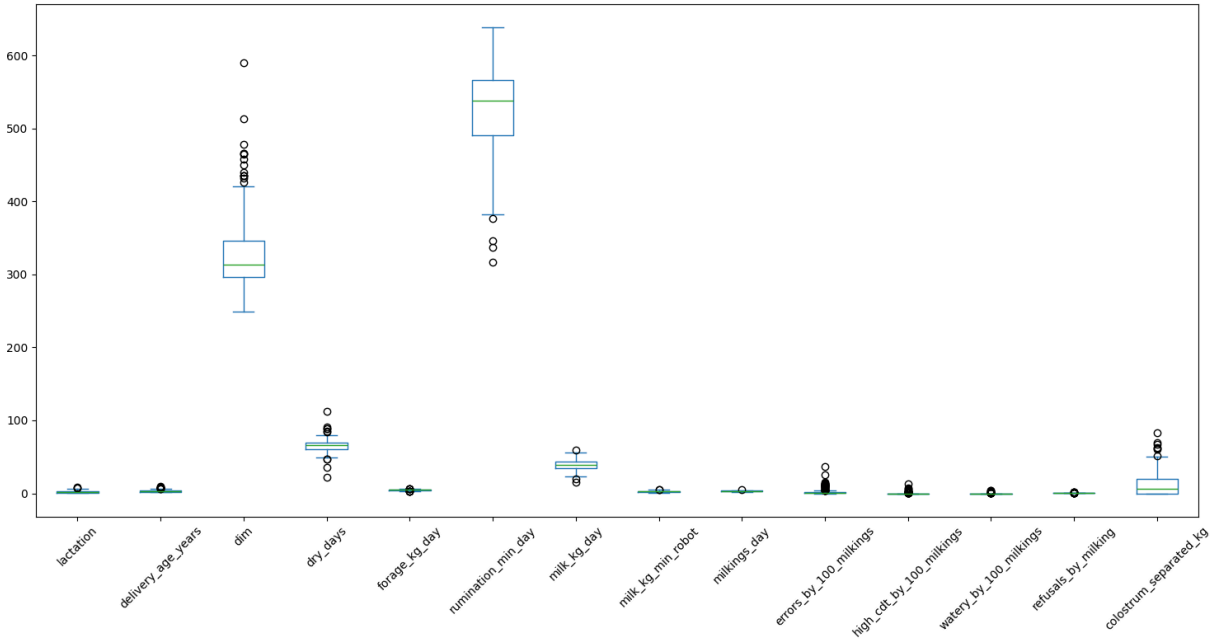
In [16]:
```
# Check the missing values per feature in the train dataset
train.isna().sum()
```

Out[16]:
```
lactation                    0
delivery_age_years           0
dim                          0
dry_days                   147
forage_kg_day                0
rumination_min_day           0
milk_kg_day                  0
milk_kg_min_robot            0
milkings_day                 0
errors_by_100_milkings       0
high_cdt_by_100_milkings     0
watery_by_100_milkings       0
refusals_by_milking          0
colostrum_separated_kg       0
dtype: int64
```

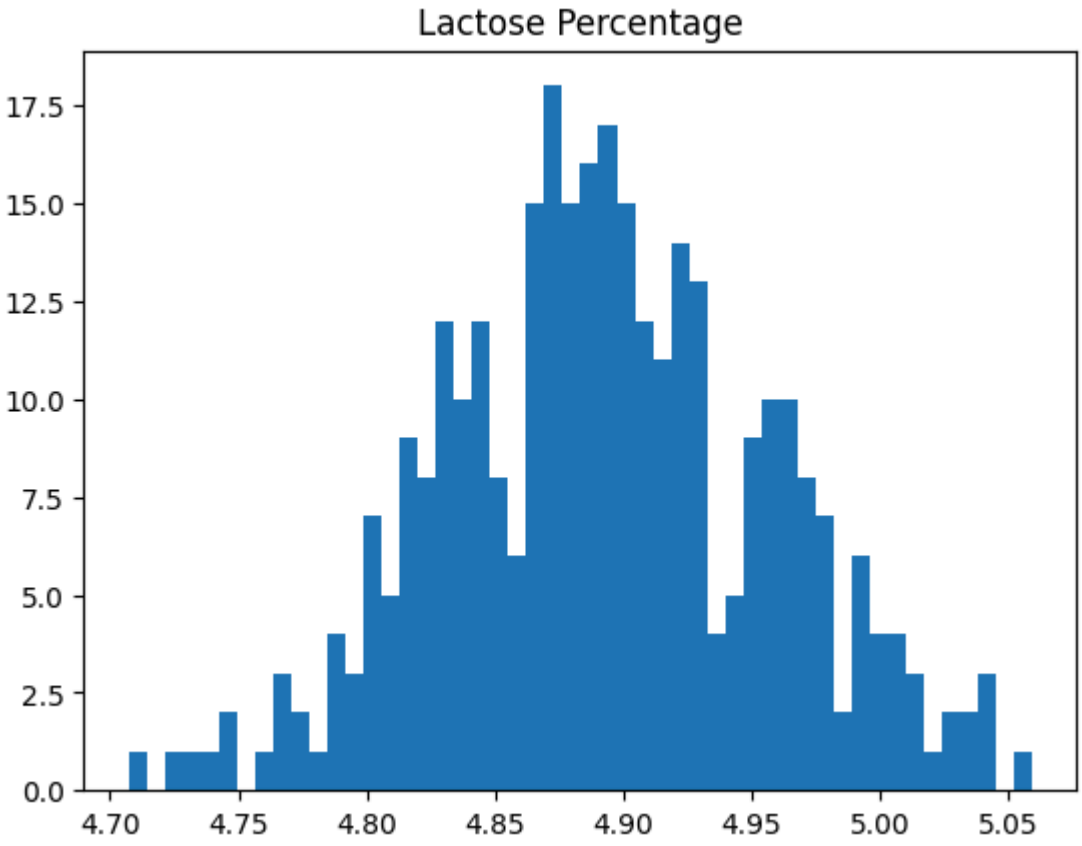
In [17]:
```
# Check the histograms of the features in the train dataset
train.hist(figsize=(15, 15), bins=50)
plt.grid(False)
plt.tight_layout()
plt.show()
```



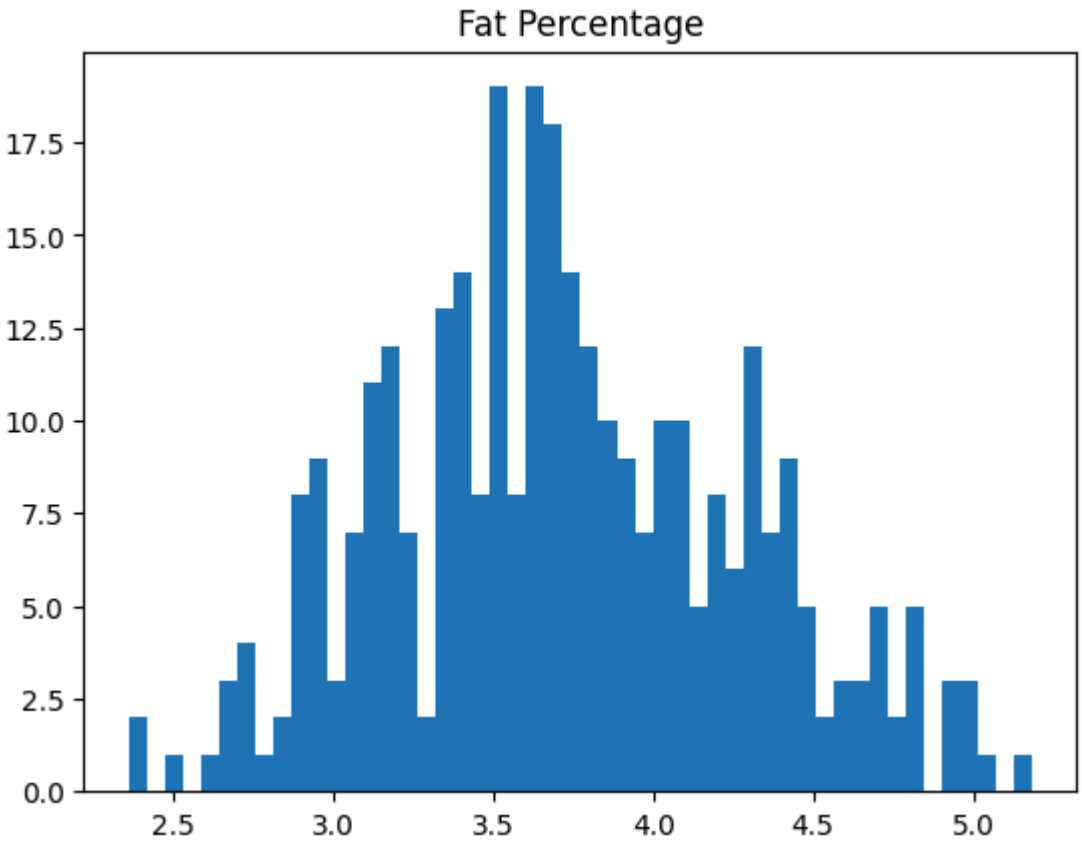
In [18]:
```
# Check the boxplots of the features in the train dataset, for possible outlier ide
train.plot(kind='box', figsize=(15, 8))
plt.xticks(ticks=range(1, len(train.columns) + 1), labels=train.columns, rotation=4
plt.tight_layout()
plt.show()
```



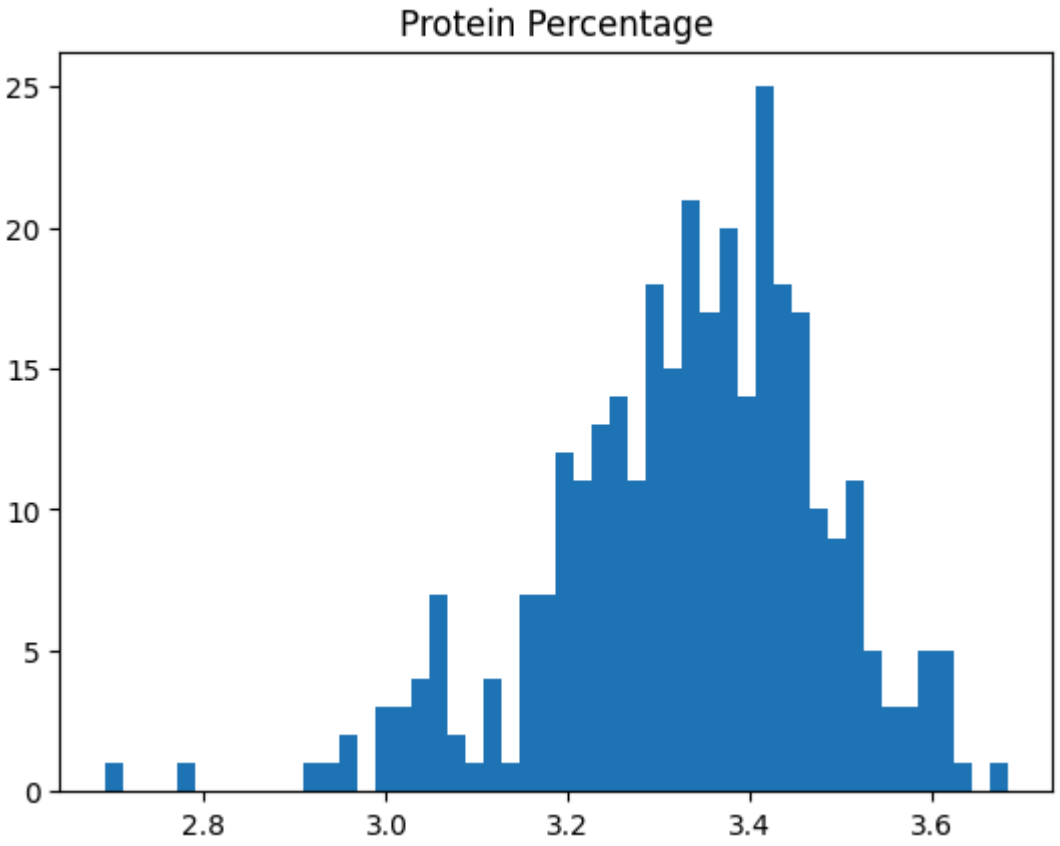
In [19]:
```
# Check the distribution of the lactose percentage target
plt.hist(y_lactose, bins=50, color='tab:blue')
plt.title('Lactose Percentage')
plt.show()
```



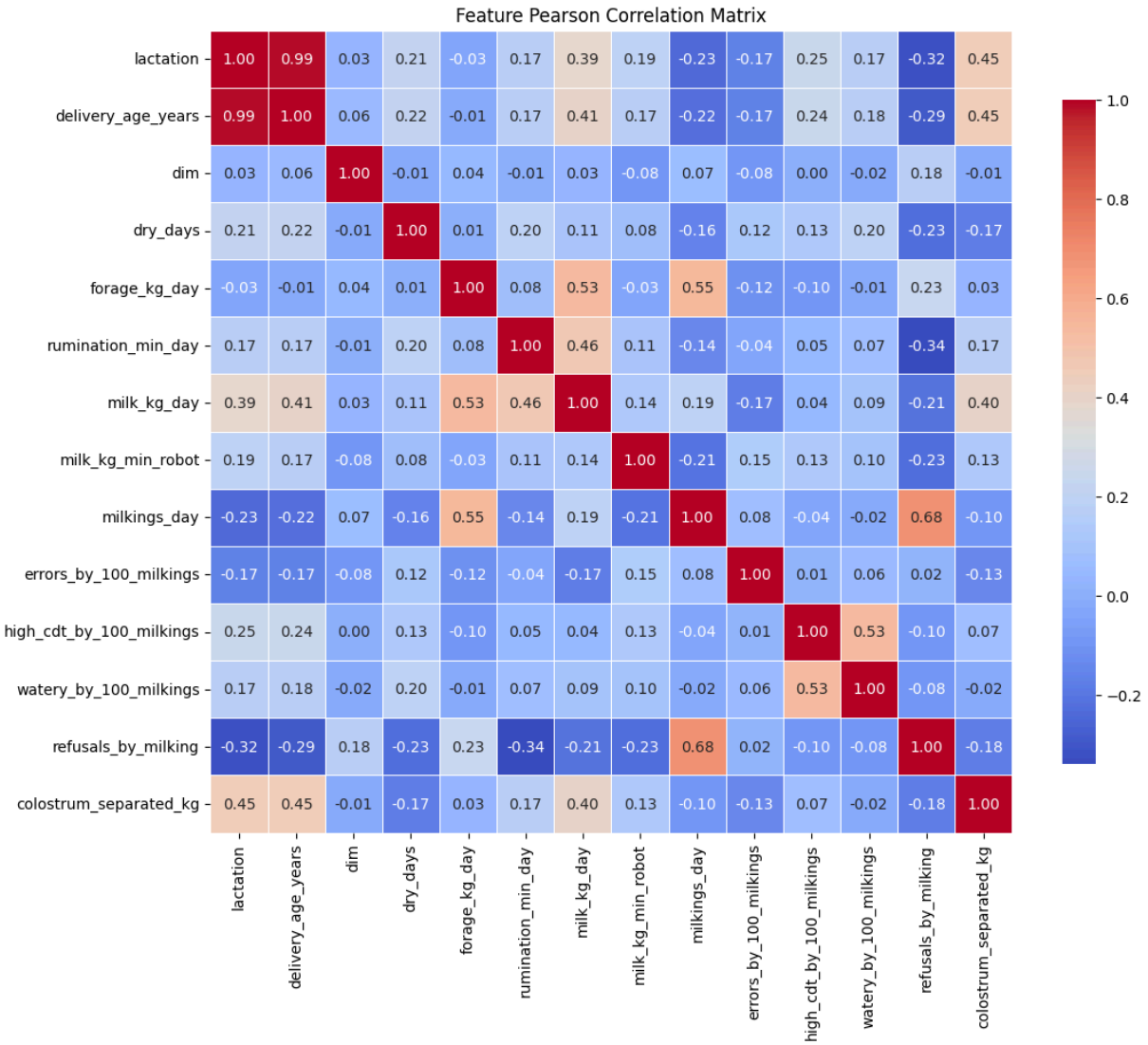
In [20]:
```
# Check the distribution of the fat percentage target
plt.hist(y_fat, bins=50, color='tab:blue')
plt.title('Fat Percentage')
plt.show()
```



In [21]:
```
# Check the distribution of the protein percentage target
plt.hist(y_protein, bins=50, color='tab:blue')
plt.title('Protein Percentage')
plt.show()
```

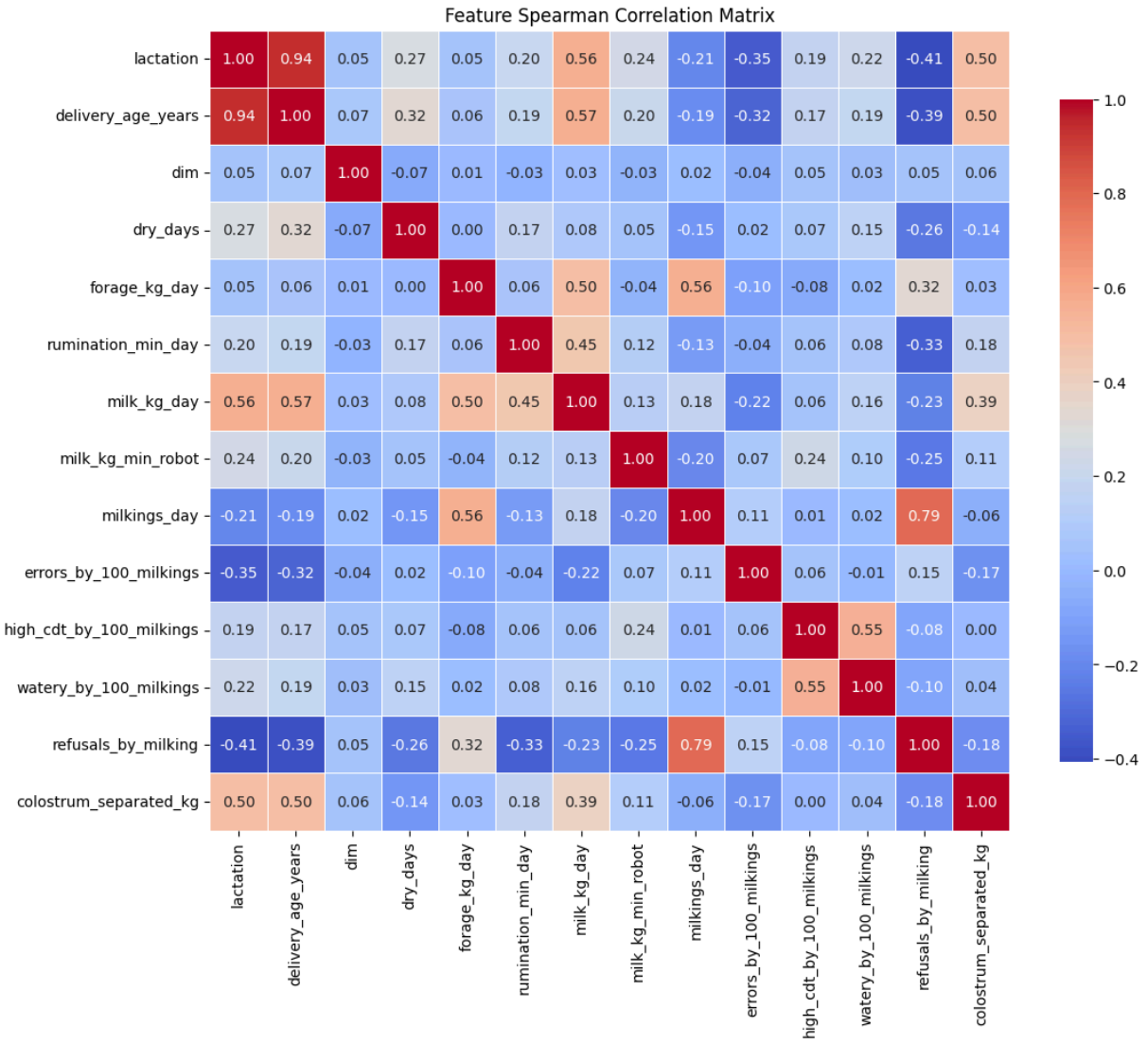


In [22]:
```
# Calculate the Pearson correlation matrix between features in the train dataset
corr_matrix = train.corr()

plt.figure(figsize=(12, 10))

sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm',
            square=True, linewidths=.5, cbar_kws={"shrink": .8})

plt.title('Feature Pearson Correlation Matrix')
plt.show()
```



In [23]:

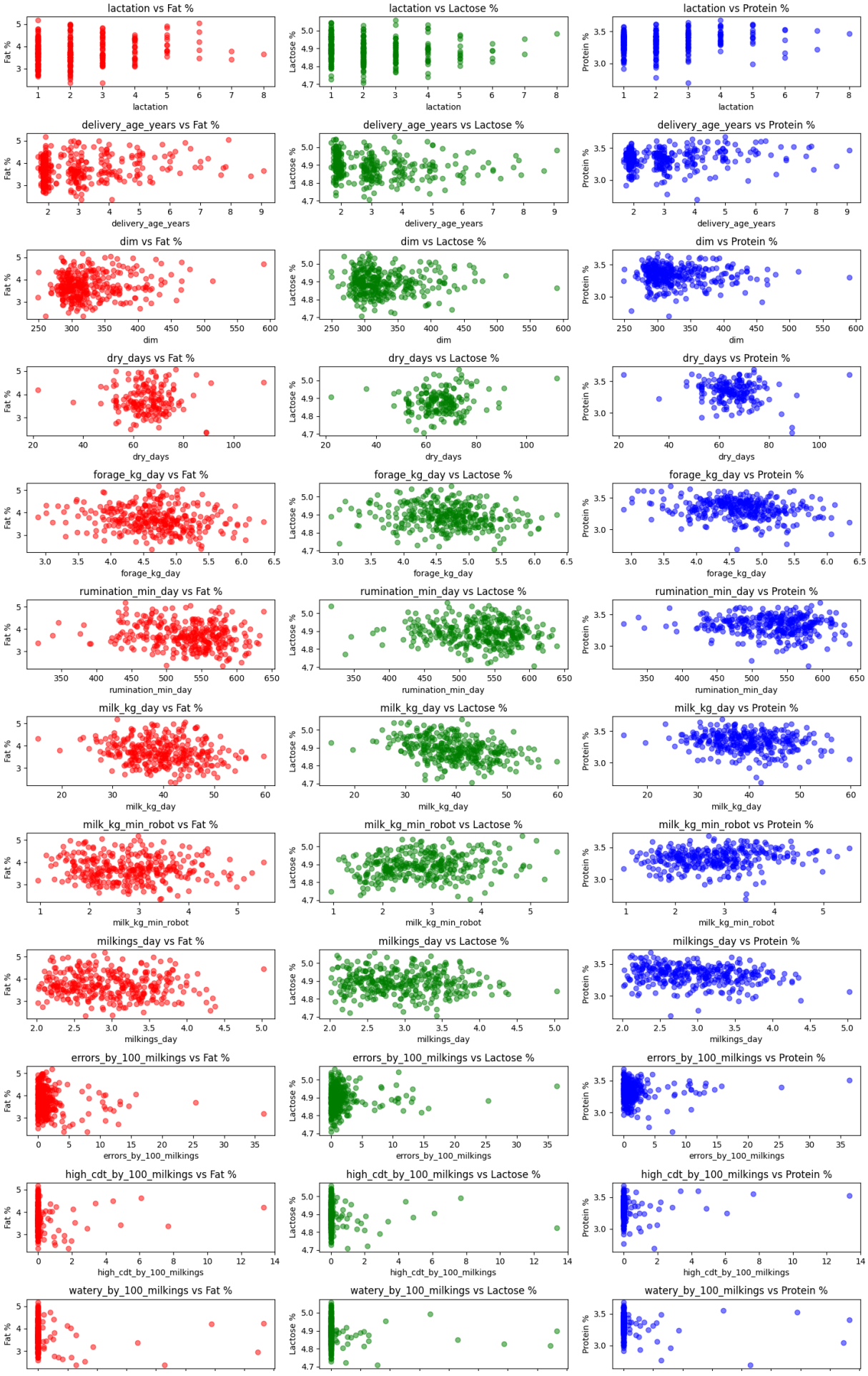
```
# Calculate the Pearson correlation matrix between features in the train dataset
corr_matrix = train.corr(method='spearman')

plt.figure(figsize=(12, 10))

sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm',
            square=True, linewidths=.5, cbar_kws={"shrink": .8})

plt.title('Feature Spearman Correlation Matrix')
plt.show()
```

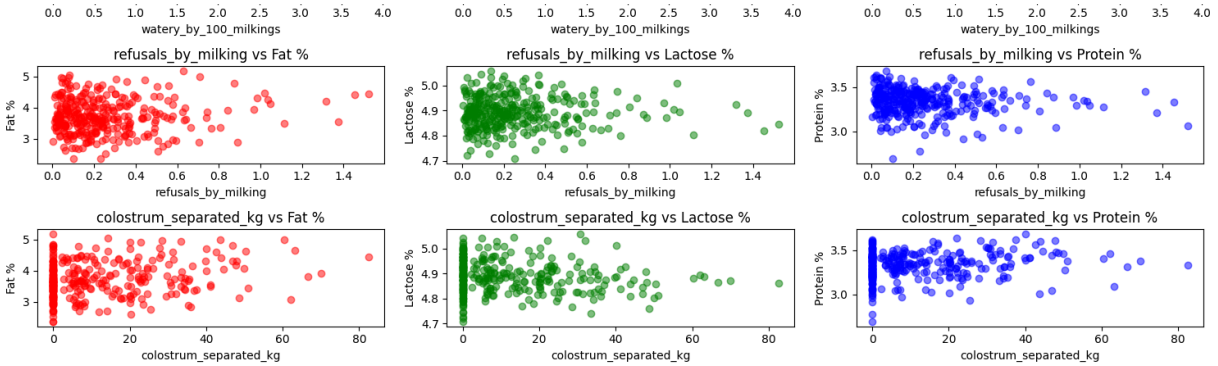


In [24]:

```
# Number of features
num_features = train.shape[1]
features = train.columns

# Create a figure with multiple subplots
fig, axes = plt.subplots(nrows=num_features, ncols=3, figsize=(15, 2 * num_features

if num_features == 1:
    axes = axes.reshape(1, -1)

# Loop through each feature and plot against each target
for i, feature in enumerate(features):

    # Scatter plot for Fat Percent
    axes[i, 0].scatter(train[feature], y_fat['fat_percent'], color='r', alpha=0.5)
    axes[i, 0].set_title(f'{feature} vs Fat %')
    axes[i, 0].set_xlabel(feature)
    axes[i, 0].set_ylabel('Fat %')

    # Scatter plot for Lactose Percent
    axes[i, 1].scatter(train[feature], y_lactose['lactose_percent'], color='g', alp
    axes[i, 1].set_title(f'{feature} vs Lactose %')
```

```
    axes[i, 1].set_xlabel(feature)
    axes[i, 1].set_ylabel('Lactose %')

    # Scatter plot for Protein Percent
    axes[i, 2].scatter(train[feature], y_protein['protein_percent'], color='b', alp
    axes[i, 2].set_title(f'{feature} vs Protein %')
    axes[i, 2].set_xlabel(feature)
    axes[i, 2].set_ylabel('Protein %')

plt.tight_layout()
plt.show()
```


lactation vs Fat %
lactation vs Lactose %
lactation vs Protein %
delivery_age_years vs Fat %
delivery_age_years vs Lactose %
delivery_age_years vs Protein %
dim vs Fat %
dim vs Lactose %
dim vs Protein %
dry_days vs Fat %
dry_days vs Lactose %
dry_days vs Protein %
forage_kg_day vs Fat %
forage_kg_day vs Lactose %
forage_kg_day vs Protein %
rumination_min_day vs Fat %
rumination_min_day vs Lactose %
rumination_min_day vs Protein %
milk_kg_day vs Fat %
milk_kg_day vs Lactose %
milk_kg_day vs Protein %
milk_kg_min_robot vs Fat %
milk_kg_min_robot vs Lactose %
milk_kg_min_robot vs Protein %
milkings_day vs Fat %
milkings_day vs Lactose %
milkings_day vs Protein %
errors_by_100_milkings vs Fat %
errors_by_100_milkings vs Lactose %
errors_by_100_milkings vs Protein %
high_cdt_by_100_milkings vs Fat %
high_cdt_by_100_milkings vs Lactose %
high_cdt_by_100_milkings vs Protein %
watery_by_100_milkings vs Fat %
watery_by_100_milkings vs Lactose %
watery_by_100_milkings vs Protein %
Fat %
Lactose %
Protein %



2. Data Preprocessing

In [25]:
```
# Dropping the records where "lactation" = 1, so that is possible to use the 'dry_d
train = train[train['lactation'] != 1]
```

In [26]:
```
# Matching the existing indexes in the target dataset
y_lactose = y_lactose.loc[train.index]
```

In [27]:
```
# Resetting the index in both train and target datasets, dropping the extra created
train.reset_index(inplace=True, drop=True)
y_lactose.reset_index(inplace=True, drop=True)
```

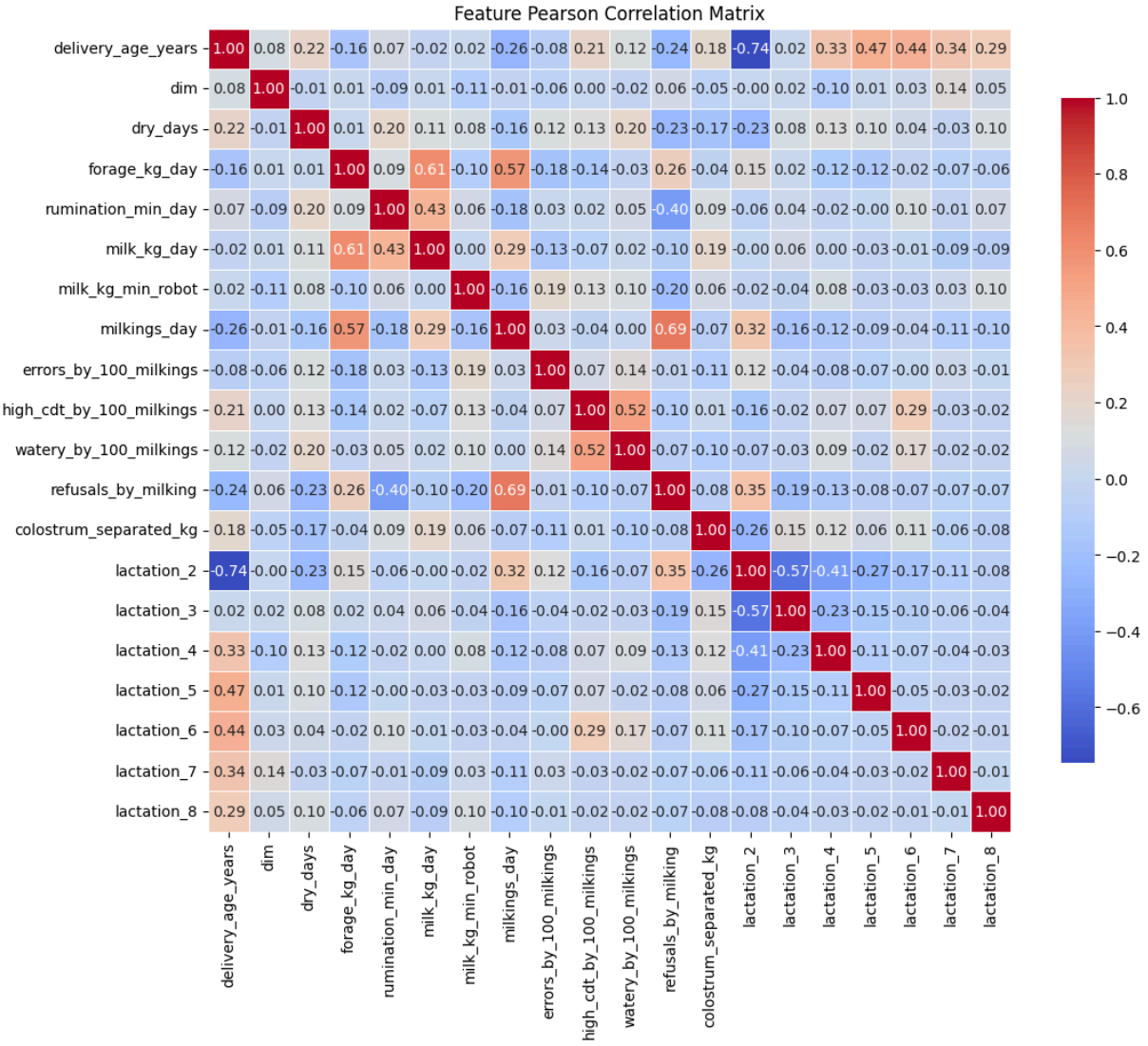
In [28]:
```
# One-hot encoding the 'lactation' feature since it is a categorical feature
train = pd.get_dummies(data=train, columns=['lactation'], dtype=int)
```

In [29]:
```
# Check the first 10 rows of the train dataset
train.head(10)
```

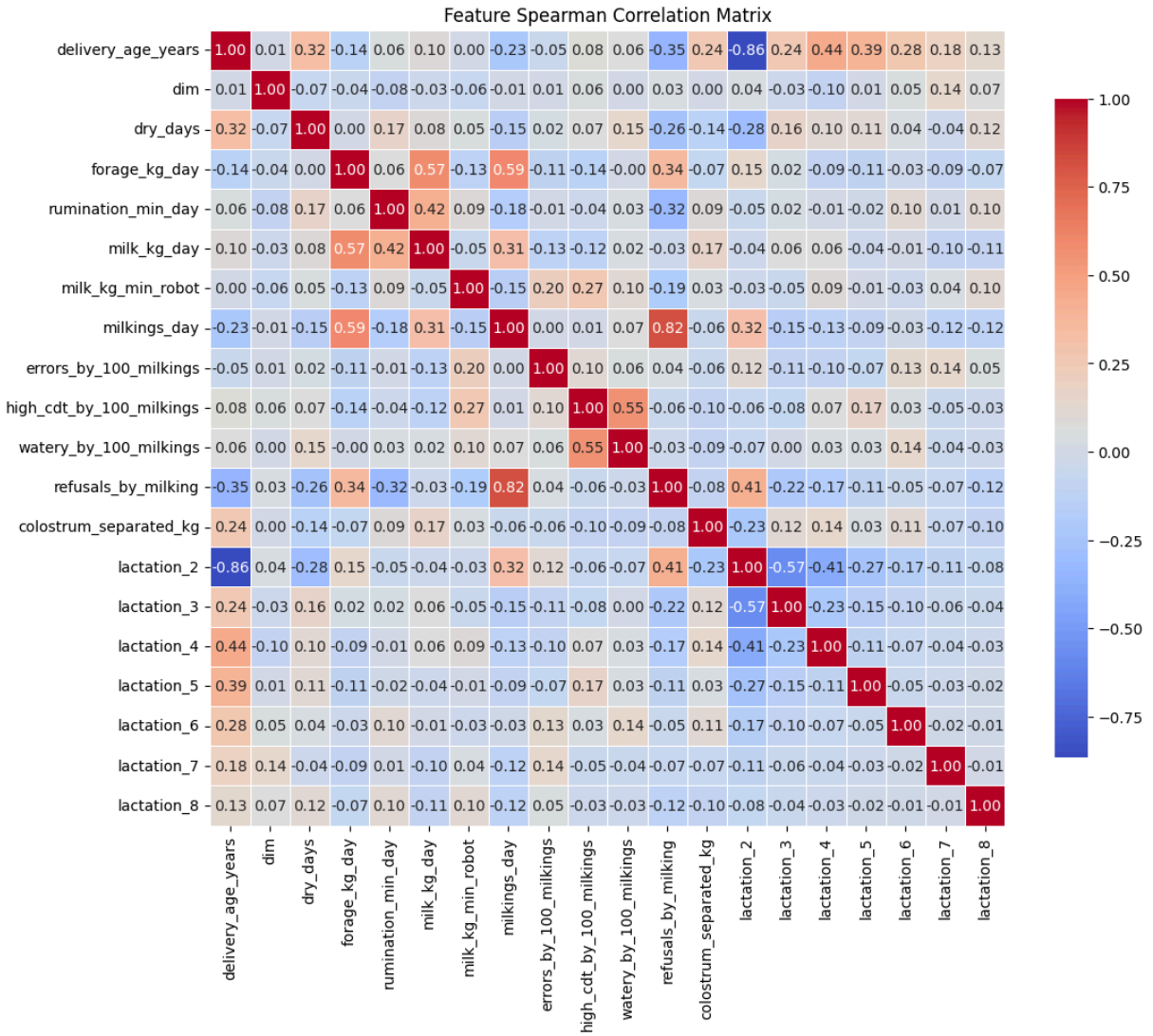
Out[29]:

	delivery_age_years	dim	dry_days	forage_kg_day	rumination_min_day	milk_kg_day	mi
0	7.750000	414	56.0	4.310918	434.814010	34.082367	
1	9.083333	357	78.0	4.167087	589.500000	36.170868	
2	7.666667	315	69.0	4.903333	542.577778	43.371746	
3	8.666667	362	69.0	4.294724	628.371901	41.683149	
4	6.500000	427	58.0	4.798618	479.334112	37.916393	
5	7.916667	359	77.0	5.092646	534.011111	44.288022	
6	6.833333	320	69.0	4.858062	543.697819	45.443438	
7	7.833333	336	58.0	4.478661	610.554896	45.106845	
8	5.833333	373	71.0	5.324906	434.320856	42.970777	
9	7.000000	298	62.0	5.680805	556.604027	47.485570	

In [30]:
```
# Calculate the Pearson correlation matrix between features in the train dataset
corr_matrix = train.corr()

plt.figure(figsize=(12, 10))

sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm',
            square=True, linewidths=.5, cbar_kws={"shrink": .8})

plt.title('Feature Pearson Correlation Matrix')
plt.show()
```



In [31]:
```
# Calculate the Spearman correlation matrix between features in the train dataset
corr_matrix = train.corr(method='spearman')

plt.figure(figsize=(12, 10))

sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm',
            square=True, linewidths=.5, cbar_kws={"shrink": .8})

plt.title('Feature Spearman Correlation Matrix')
plt.show()
```



In [32]:
```
# Dropping the 'dim' feature, since it has insignificant correlations (Pearson and
train.drop('dim', axis=1, inplace=True)
```

In [33]:
```
train.to_csv("train_preprocessed.csv", index = False)
```

In [34]:
```
print(f'There are {train.duplicated().sum()} duplicated records.')
```

```
There are 0 duplicated records.
```

3. Genetic Programming

GP with single tournament selection

Using max_init_depth = 3:

In [35]:
```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values
```

```
# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}


def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre

        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
```

```
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=grow,
            selector=prm_tournament(pressure=gp_kwargs['selection_pressure']),
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [36]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_single_tournament_grow_max_init_3 = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state=42
)

# Append the results of the function call to the list
results_gp_single_tournament_grow_max_init_3.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 25
```

```
)

# Append the results of the function call to the list
results_gp_single_tournament_grow_max_init_3.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state=63
)

# Append the results of the function call to the list
results_gp_single_tournament_grow_max_init_3.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gp_single_tournament_grow_max_init_3 = list(itertools.chain.from_iterable(r

print("Cross-validation scores for all runs:", results_gp_single_tournament_grow_ma
print("Mean cross-validation score for all runs:", np.mean(results_gp_single_tourna
```

```
Cross-validation scores for run 1: [8.4400177e-05 5.8917999e-03 4.0674207e-04 4.0674
207e-04 8.4400177e-05
 4.0674207e-04 4.0674207e-04 9.8224944e-03 4.0674207e-04 9.9515915e-04]
Mean cross-validation score: 0.0018911965
Cross-validation scores for run 2: [8.4400177e-05 8.4400177e-05 6.3371658e-04 4.0674
207e-04 8.4400177e-05
 3.8828850e-03 6.2036508e-04 6.2036508e-04 8.4400177e-05 6.3371658e-04]
Mean cross-validation score: 0.00071353914
Cross-validation scores for run 3: [9.9515915e-04 8.4400177e-05 8.4400177e-05 1.0375
977e-03 2.6464462e-04
 5.1579475e-03 2.6793480e-03 5.9006615e-03 8.4400177e-05 1.2021065e-03]
Mean cross-validation score: 0.0017490666
Cross-validation scores for all runs: [8.440018e-05, 0.0058918, 0.00040674207, 0.000
40674207, 8.440018e-05, 0.00040674207, 0.00040674207, 0.009822494, 0.00040674207, 0.
0009951591, 8.440018e-05, 8.440018e-05, 0.0006337166, 0.00040674207, 8.440018e-05,
0.003882885, 0.0006203651, 0.0006203651, 8.440018e-05, 0.0006337166, 0.0009951591,
8.440018e-05, 8.440018e-05, 0.0010375977, 0.00026464462, 0.0051579475, 0.002679348,
0.0059006615, 8.440018e-05, 0.0012021065]
Mean cross-validation score for all runs: 0.0014512674
```

Using max_init_depth = 5:

In [37]:

```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 5,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}

def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre

        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
```

```
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=grow,
            selector=prm_tournament(pressure=gp_kwargs['selection_pressure']),
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [38]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_single_tournament_grow_max_init_5 = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state=42
)

# Append the results of the function call to the list
results_gp_single_tournament_grow_max_init_5.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state=25
```

```
)

# Append the results of the function call to the list
results_gp_single_tournament_grow_max_init_5.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state=63
)

# Append the results of the function call to the list
results_gp_single_tournament_grow_max_init_5.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gp_single_tournament_grow_max_init_5 = list(itertools.chain.from_iterable(r

print("Cross-validation scores for all runs:", results_gp_single_tournament_grow_ma
print("Mean cross-validation score for all runs:", np.mean(results_gp_single_tourna
```

```
Cross-validation scores for run 1: [0.00485849 0.08602858 0.00380421 0.00777578 0.00
485849 0.00805859
 3.202504   0.01914341 0.00734425 1.562469  ]
Mean cross-validation score: 0.49068445
Cross-validation scores for run 2: [0.01049538 0.00097942 0.01321173 0.00185919 0.01
744332 0.00734425
 0.0010376  0.00860317 0.16542989 0.00485849]
Mean cross-validation score: 0.023126243
Cross-validation scores for run 3: [0.00485849 0.00485849 0.00485849 0.00485849 0.00
734425 0.00734425
 0.01795318 0.00485849 0.00734425 0.00734425]
Mean cross-validation score: 0.007162263
Cross-validation scores for all runs: [0.004858494, 0.086028576, 0.0038042066, 0.007
77578, 0.004858494, 0.008058589, 3.202504, 0.019143414, 0.0073442464, 1.562469, 0.01
0495383, 0.0009794235, 0.013211727, 0.001859188, 0.017443316, 0.0073442464, 0.001037
5977, 0.008603168, 0.16542989, 0.004858494, 0.004858494, 0.004858494, 0.004858494,
0.004858494, 0.0073442464, 0.0073442464, 0.01795318, 0.004858494, 0.0073442464, 0.00
73442464]
Mean cross-validation score for all runs: 0.17365767
```

Our **best model** is the first one (**max_init_depth = 3**), so it will be used to test other initializers:

RHH as initializer:

In [39]:
```
# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}

def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre
```

```
        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=rhh,
            selector=prm_tournament(pressure=gp_kwargs['selection_pressure']),
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [40]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_single_tournament_rhh = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 42
)
results_gp_single_tournament_rhh.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
```

```
    shuffle=True,
    **gp_params,
    random_state = 25
)

results_gp_single_tournament_rhh.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 63
)

results_gp_single_tournament_rhh.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gp_single_tournament_rhh = list(itertools.chain.from_iterable(results_gp_si

print("Cross-validation scores for all runs:", results_gp_single_tournament_rhh)
print("Mean cross-validation score for all runs:", np.mean(results_gp_single_tourna
```

```
Cross-validation scores for run 1: [0.00734425 0.11675034 0.00485849 0.05249421 0.00
485849 0.02351561
 0.00495577 0.00734425 0.01059628 0.00639057]
Mean cross-validation score: 0.023910824
Cross-validation scores for run 2: [0.00734425 0.00485849 0.00485849 0.02169398 0.00
485849 0.01868325
 0.01181546 0.06952292 0.00342321 0.00341516]
Mean cross-validation score: 0.015047371
Cross-validation scores for run 3: [0.00197536 0.00485849 0.00485849 0.00734425 0.01
728839 0.0179647
 0.01128802 0.01086449 0.00826073 0.00485849]
Mean cross-validation score: 0.00895614
Cross-validation scores for all runs: [0.0073442464, 0.11675034, 0.004858494, 0.0524
94206, 0.004858494, 0.023515606, 0.004955769, 0.0073442464, 0.010596276, 0.006390571
6, 0.0073442464, 0.004858494, 0.004858494, 0.02169398, 0.004858494, 0.018683251, 0.0
11815458, 0.06952292, 0.003423214, 0.003415161, 0.001975359, 0.004858494, 0.00485849
4, 0.0073442464, 0.01728839, 0.017964698, 0.011288017, 0.010864487, 0.008260727, 0.0
04858494]
Mean cross-validation score for all runs: 0.015971445
```

FULL initialization:

In [41]:

```
# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}

def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42, **g
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre

        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
            n_jobs=8
```

```
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=full,
            selector=prm_tournament(pressure=gp_kwargs['selection_pressure']),
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [42]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_single_tournament_full = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 42
)

results_gp_single_tournament_full.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 25
)
```

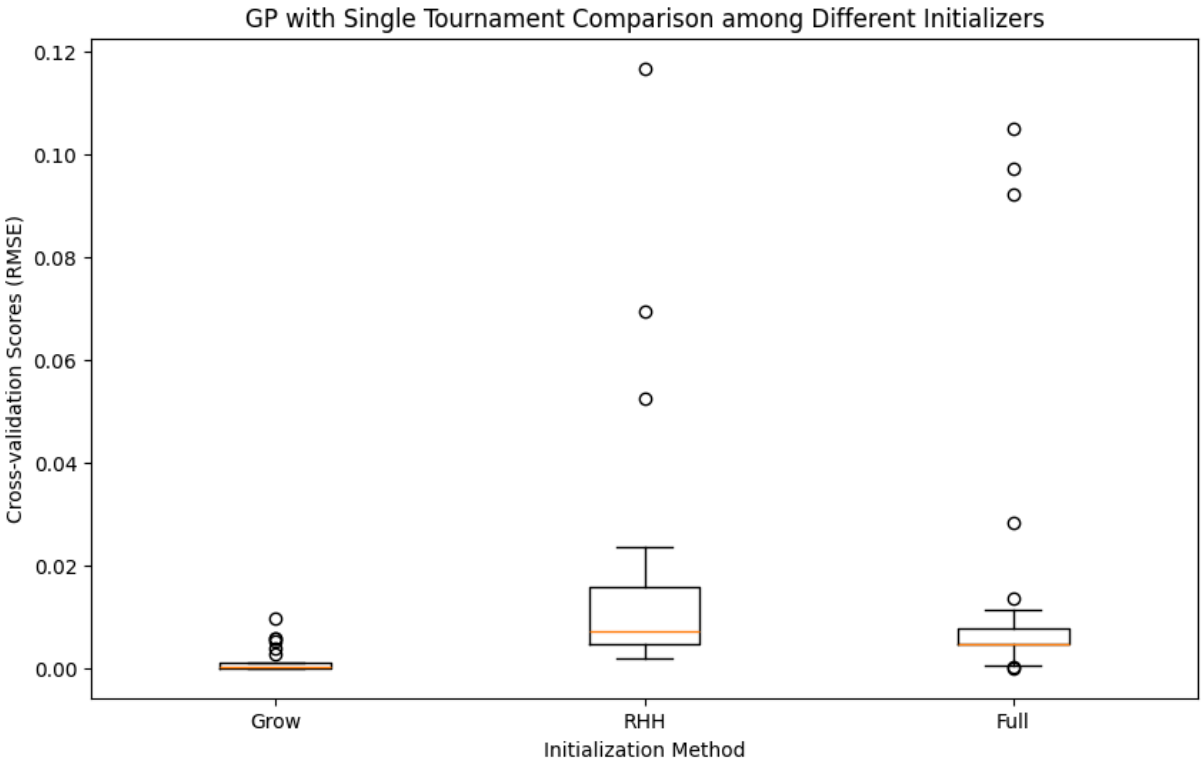
```
results_gp_single_tournament_full.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 63
)

results_gp_single_tournament_full.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gp_single_tournament_full = list(itertools.chain.from_iterable(results_gp_s

print("Cross-validation scores for all runs:", results_gp_single_tournament_full)
print("Mean cross-validation score for all runs:", np.mean(results_gp_single_tourna
```

```
Cross-validation scores for run 1: [0.01359452 0.02842383 0.00012875 0.00782585 0.00
485849 0.01136827
 0.00485849 0.00734425 0.00067377 0.01059628]
Mean cross-validation score: 0.008967251
Cross-validation scores for run 2: [0.00485849 0.0019422  0.00485849 0.00485849 0.00
485849 0.00039673
 0.00734425 0.00485849 0.10493678 0.0011797 ]
Mean cross-validation score: 0.014009212
Cross-validation scores for run 3: [0.00485849 0.00485849 0.00603999 0.00734425 0.09
71899  0.00485849
 0.00782585 0.00485849 0.09224116 0.00485849]
Mean cross-validation score: 0.02349336
Cross-validation scores for all runs: [0.013594525, 0.028423833, 0.00012874603, 0.00
7825851, 0.004858494, 0.011368275, 0.004858494, 0.0073442464, 0.00067377085, 0.01059
6276, 0.004858494, 0.0019421994, 0.004858494, 0.004858494, 0.004858494, 0.0003967285
2, 0.0073442464, 0.004858494, 0.10493678, 0.001179695, 0.004858494, 0.004858494, 0.0
060399882, 0.0073442464, 0.0971899, 0.004858494, 0.007825851, 0.004858494, 0.0922411
6, 0.004858494]
Mean cross-validation score for all runs: 0.015489941
```

BEST SCORE: GROW initialization

In [43]:
```
# Combine the scores into a list
data = [results_gp_single_tournament_grow_max_init_3, results_gp_single_tournament_

# Labels for each method
```

```
labels = ['Grow', 'RHH', 'Full']

# Create the boxplot
plt.figure(figsize=(10, 6))
plt.boxplot(data, labels=labels)
plt.title('GP with Single Tournament Comparison among Different Initializers')
plt.ylabel('Cross-validation Scores (RMSE)')
plt.xlabel('Initialization Method')
plt.grid(False)
plt.show()
```



Double Tournament Selection

> **Double Tournament Selection Implementation**

This was implemented directly inside gpolnel, in `selectors.py`. The code in the next cell shows our work.

In [44]:
```
# def prm_double_tournament(pressure, size_criteria=True):
#     """
#     Implements a double tournament selection based on fitness and tree size.

#     Parameters:
#     - pressure: float, the proportion of the population to sample for each tourna
#     - size_criteria: bool, True if the final selection is based on tree size, Fal

#     Returns:
#     - Function that performs the double tournament on a given population.
#     """
```

```
#     def double_tournament(pop, min_=True):
#         """
#         Conducts two sequential tournaments to select an individual from the popu

#         Parameters:
#         - pop: Population from which to select individuals.
#         - min_: Boolean, True if minimizing fitness, False if maximizing.

#         Returns:
#         - Index of the winning individual based on the criteria.
#         """

#         def get_size(individual):
#             """
#             Returns the size of the individual.
#             Uses get_size method if available, otherwise uses the length of the r
#             """
#             if isinstance(individual, Tree):
#                 return individual.get_size()
#             return len(individual.repr_)

#         def tournament(pop, fitness_min=True):
#             """
#             Conducts a single tournament based on fitness.

#             Parameters:
#             - pop: Population from which to select individuals.
#             - fitness_min: Boolean, True if minimizing fitness, False if maximizi

#             Returns:
#             - Index of the winner of the tournament.
#             """
#             pool_size = int(len(pop.individuals) * pressure)
#             contestants = random.sample(pop.individuals, pool_size)
#             if fitness_min:
#                 winner = min(contestants, key=lambda ind: ind.fit)
#             else:
#                 winner = max(contestants, key=lambda ind: ind.fit)
#             return pop.individuals.index(winner)

#         # First stage: Perform two independent tournaments based on fitness
#         winner1_idx = tournament(pop, fitness_min=min_)
#         winner2_idx = tournament(pop, fitness_min=min_)

#         # Second stage: Compete based on tree size or fitness
#         winner1 = pop.individuals[winner1_idx]
#         winner2 = pop.individuals[winner2_idx]

#         if size_criteria:
#             final_winner = min([winner1, winner2], key=get_size)
#         else:
#             final_winner = min([winner1, winner2], key=lambda ind: ind.fit) if mi

#         return pop.individuals.index(final_winner)

#     return double_tournament
```

GP With Double Tournament Selection, Grow Initialization

In [45]:

```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}

def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre
```

```
        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=grow,
            selector=prm_double_tournament(pressure=gp_kwargs['selection_pressure']
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [46]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_double_tournament_grow = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 42
)

results_gp_double_tournament_grow.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score for run 1:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
```

```
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 25
)

results_gp_double_tournament_grow.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score for run 2:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 63
)

results_gp_double_tournament_grow.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score for run 3:", np.mean(run_3))

del run_3
gc.collect()

results_gp_double_tournament_grow = list(itertools.chain.from_iterable(results_gp_d

print("Cross-validation scores for all runs:", results_gp_double_tournament_grow)
print("Mean cross-validation score for all runs:", np.mean(results_gp_double_tourna
```

```
Cross-validation scores for run 1: [0.08439542 0.09939098 0.08439542 0.08465728 0.08
666936 0.09999358
 0.08439542 0.08666936 0.08439542 0.08598571]
Mean cross-validation score for run 1: 0.088094786
Cross-validation scores for run 2: [0.00754274 0.08939503 0.08465727 0.08666936 0.08
666936 0.08439542
 0.08603345 0.09939098 0.00119114 0.08666936]
Mean cross-validation score for run 2: 0.071261406
Cross-validation scores for run 3: [0.0842396  0.11873426 2.5946794  0.08543248 0.08
439542 0.09939098
 0.08603345 0.08598571 0.08598571 0.08791834]
Mean cross-validation score for run 3: 0.34127954
Cross-validation scores for all runs: [0.084395416, 0.09939098, 0.084395416, 0.08465
728, 0.086669356, 0.09999358, 0.084395416, 0.086669356, 0.084395416, 0.085985705, 0.
0075427354, 0.08939503, 0.084657274, 0.086669356, 0.086669356, 0.084395416, 0.086033
45, 0.09939098, 0.0011911392, 0.086669356, 0.0842396, 0.118734255, 2.5946794, 0.0854
32485, 0.084395416, 0.09939098, 0.08603345, 0.085985705, 0.085985705, 0.08791834]
Mean cross-validation score for all runs: 0.16687857
```

GP With Double Tournament Selection, RHH Initialization

In [47]:

```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}

def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre
```

```
        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=rhh,
            selector=prm_double_tournament(pressure=gp_kwargs['selection_pressure']
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [48]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_double_tournament_rhh = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 42
)

results_gp_double_tournament_rhh.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score for run 1:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
    y=y,
```

```
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 25
)

results_gp_double_tournament_rhh.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score for run 2:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 63
)

results_gp_double_tournament_rhh.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score for run 3:", np.mean(run_3))

del run_3
gc.collect()

results_gp_double_tournament_rhh = list(itertools.chain.from_iterable(results_gp_do

print("Cross-validation scores for all runs:", results_gp_double_tournament_rhh)
print("Mean cross-validation score for all runs:", np.mean(results_gp_double_tourna
```

```
Cross-validation scores for run 1: [1.2567044e-02 6.5040588e-04 4.5637719e-02 1.4680
892e-01 2.7202606e-02
 4.8584938e-03 8.8167191e-04 2.5796835e+00 4.8584938e-03 4.5338325e-02]
Mean cross-validation score for run 1: 0.28684872
Cross-validation scores for run 2: [0.6274741  0.0147213  0.00782585 0.00734425 0.00
734425 0.07196867
 0.00734425 0.05123913 0.00253534 0.01687193]
Mean cross-validation score for run 2: 0.081466906
Cross-validation scores for run 3: [1.1633019e-02 4.8584938e-03 1.1075974e-01 7.3442
464e-03 2.5796835e+00
 6.1388329e-02 2.5024414e-03 6.1464310e-04 5.3168651e-02 7.8258514e-03]
Mean cross-validation score for run 3: 0.28397787
Cross-validation scores for all runs: [0.012567044, 0.0006504059, 0.04563772, 0.1468
0892, 0.027202606, 0.004858494, 0.0008816719, 2.5796835, 0.004858494, 0.045338325,
0.6274741, 0.014721299, 0.007825851, 0.0073442464, 0.0073442464, 0.071968675, 0.0073
442464, 0.051239125, 0.002535343, 0.01687193, 0.011633019, 0.004858494, 0.11075974,
0.0073442464, 2.5796835, 0.06138833, 0.0025024414, 0.0006146431, 0.05316865, 0.00782
5851]
Mean cross-validation score for all runs: 0.21743117
```

GP With Double Tournament Selection, Full Initialization

In [49]:

```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,  # Since we are using the entire dataset in each fold
    'device': 'cpu'
}

gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}

def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre
```

```
        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=full,
            selector=prm_double_tournament(pressure=gp_kwargs['selection_pressure']
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_kwargs['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

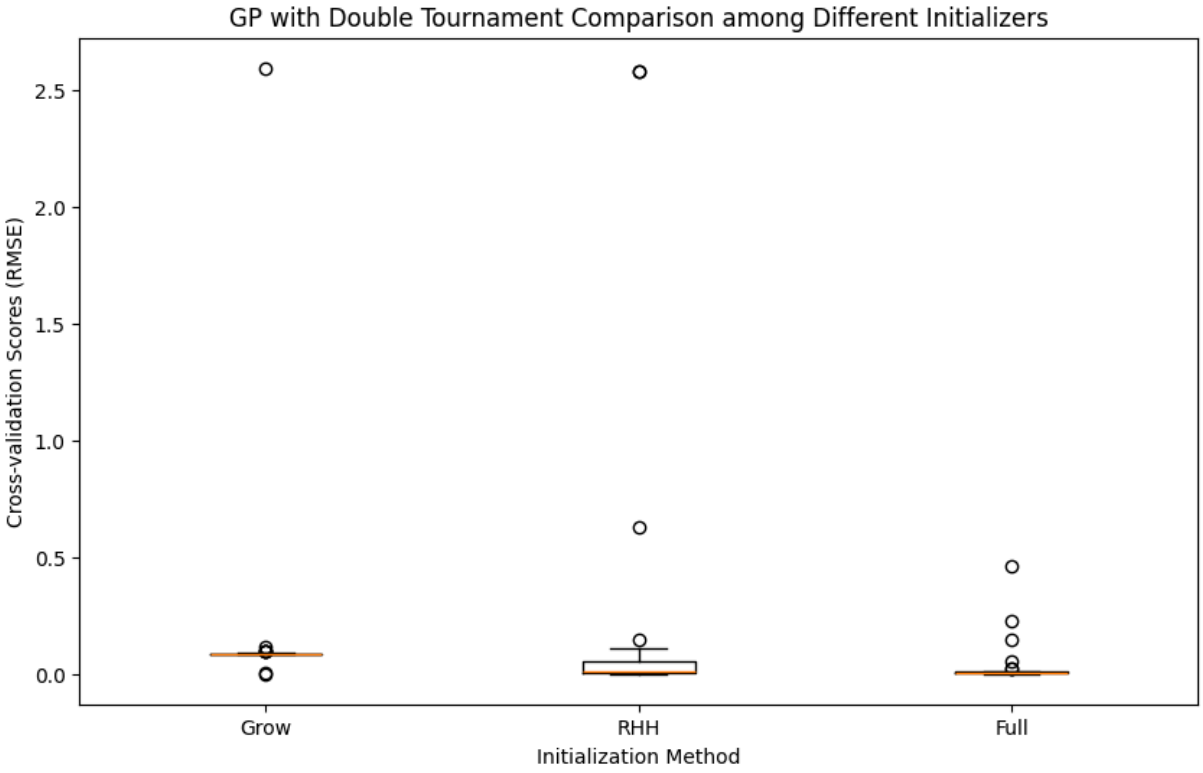
In [50]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gp_double_tournament_full = []

run_1 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 42
)

results_gp_double_tournament_full.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score for run 1:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_genetic_programming(
    X=X,
    y=y,
```

```
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 25
)

results_gp_double_tournament_full.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score for run 2:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state = 63
)

results_gp_double_tournament_full.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score for run 3:", np.mean(run_3))

del run_3
gc.collect()

results_gp_double_tournament_full = list(itertools.chain.from_iterable(results_gp_d

print("Cross-validation scores for all runs:", results_gp_double_tournament_full)
print("Mean cross-validation score for all runs:", np.mean(results_gp_double_tourna
```

```
Cross-validation scores for run 1: [0.00485849 0.00485849 0.00485849 0.00782585 0.00
485849 0.02232545
 0.00826073 0.00968185 0.00485849 0.02472592]
Mean cross-validation score for run 1: 0.009711226
Cross-validation scores for run 2: [0.00485849 0.00119591 0.05611753 0.00485849 0.00
485849 0.00485849
 0.14548908 0.01015257 0.4640285  0.22870684]
Mean cross-validation score for run 2: 0.09251244
Cross-validation scores for run 3: [0.00485849 0.00485849 0.00435829 0.00734425 0.00
478125 0.00485849
 0.00485849 0.00485849 0.00734425 0.00485849]
Mean cross-validation score for run 3: 0.0052978992
Cross-validation scores for all runs: [0.004858494, 0.004858494, 0.004858494, 0.0078
25851, 0.004858494, 0.022325449, 0.008260727, 0.009681854, 0.004858494, 0.024725916,
0.004858494, 0.0011959076, 0.056117535, 0.004858494, 0.004858494, 0.004858494, 0.145
48908, 0.010152566, 0.4640285, 0.22870684, 0.004858494, 0.004858494, 0.0043582916,
0.0073442464, 0.004781246, 0.004858494, 0.004858494, 0.004858494, 0.0073442464, 0.00
4858494]
Mean cross-validation score for all runs: 0.035840522
```

In [51]:
```
# Combine the scores into a list
data = [results_gp_double_tournament_grow, results_gp_double_tournament_rhh, result

# Labels for each method
labels = ['Grow', 'RHH', 'Full']

# Create the boxplot
plt.figure(figsize=(10, 6))
plt.boxplot(data, labels=labels)
plt.title('GP with Double Tournament Comparison among Different Initializers')
plt.ylabel('Cross-validation Scores (RMSE)')
plt.xlabel('Initialization Method')
plt.grid(False)
plt.show()
```



4. Geometric Semantic Genetic Programming

In [52]:
```
seed = 42
#device = 'cuda' if torch.cuda.is_available() else 'cpu'
device = 'mps' if torch.backends.mps.is_available() else 'cpu' #para mac
total_batches = 1
```

In [53]:
```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()
```

```
# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}

gsgp_params = {
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}

#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic

        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
```

```
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=grow,
            selector=prm_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
            #only store semantic of individuals so we need new genetic algorithms
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [54]:

```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gsgsp_grow = []

run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_grow.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
```

```
gc.collect()

run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 25
)

results_gsgsp_grow.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)

results_gsgsp_grow.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_grow = list(itertools.chain.from_iterable(results_gsgsp_grow))

print("Cross-validation scores for all runs: ", results_gsgsp_grow)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_grow))
```

```
Cross-validation scores for run 1: [0.16142741 0.13936208 0.17470771 0.23570862 0.17
73558  0.21017362
 0.14266491 0.19933338 0.13897026 0.21575072]
Mean cross-validation score: 0.17954545
Cross-validation scores for run 2: [0.22148766 0.1466242  0.23574154 0.19968735 0.19
830789 0.14045523
 0.18337578 0.11007307 0.16886364 0.20820047]
Mean cross-validation score: 0.18128167
Cross-validation scores for run 3: [0.20775709 0.19781317 0.1784838  0.1154378  0.17
786083 0.14163609
 0.27251914 0.1074954  0.0947182  0.22515738]
Mean cross-validation score: 0.17188789
Cross-validation scores for all runs:  [0.16142741, 0.13936208, 0.17470771, 0.235708
62, 0.1773558, 0.21017362, 0.14266491, 0.19933338, 0.13897026, 0.21575072, 0.2214876
6, 0.1466242, 0.23574154, 0.19968735, 0.19830789, 0.14045523, 0.18337578, 0.11007307
5, 0.16886364, 0.20820047, 0.20775709, 0.19781317, 0.1784838, 0.1154378, 0.17786083,
0.14163609, 0.27251914, 0.1074954, 0.0947182, 0.22515738]
Mean cross-validation score for all runs:  0.17757164
```

Changing parameters to optimize slightly:

p_constants = 0.4

max_init_depth = 2

selection_pressure = 0.2

mutation_prob = 0.3

In [55]:

```python
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.4,
    'max_init_depth': 2,
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}

gsgp_params = {
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.2,
    'mutation_prob': 0.3,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}
```

```
#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,

    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic

        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=grow,
            selector=prm_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
```

```
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [56]:

```
# Initialize the results list
results_gsgsp_grow_optimized = []

# Perform K-Fold cross-validation with the genetic programming algorithm
run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_grow_optimized.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

# Perform K-Fold cross-validation with the genetic programming algorithm
run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 25
)

results_gsgsp_grow_optimized.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

# Perform K-Fold cross-validation with the genetic programming algorithm
run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)
```

```
results_gsgsp_grow_optimized.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_grow_optimized = list(itertools.chain.from_iterable(results_gsgsp_gro

print("Cross-validation scores for all runs: ", results_gsgsp_grow_optimized)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_grow_opti
```

```
Cross-validation scores for run 1: [0.06919065 0.06820968 0.06319376 0.06295585 0.06
194712 0.13253559
 0.06664439 0.14691636 0.06278697 0.16000445]
Mean cross-validation score: 0.08943848
Cross-validation scores for run 2: [0.14582399 0.07033191 0.14352673 0.06132867 0.07
010219 0.06211182
 0.13852659 0.14321253 0.14566353 0.16241285]
Mean cross-validation score: 0.11430408
Cross-validation scores for run 3: [0.14658622 0.06775679 0.14428812 0.06917781 0.06
427719 0.06740413
 0.06134895 0.15870939 0.15442991 0.06553341]
Mean cross-validation score: 0.09995119
Cross-validation scores for all runs:  [0.06919065, 0.06820968, 0.06319376, 0.062955
85, 0.06194712, 0.13253559, 0.066644385, 0.14691636, 0.06278697, 0.16000445, 0.14582
399, 0.07033191, 0.14352673, 0.061328672, 0.070102185, 0.062111825, 0.13852659, 0.14
321253, 0.14566353, 0.16241285, 0.14658622, 0.067756794, 0.14428812, 0.069177814, 0.
06427719, 0.06740413, 0.061348952, 0.15870939, 0.15442991, 0.065533414]
Mean cross-validation score for all runs:  0.10123125
```

Applying other initializations:

RHH initializations:

In [57]:
```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.4,
    'max_init_depth': 2,
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}

gsgp_params = {
```

```
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.2,
    'mutation_prob': 0.3,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}

#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,

    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic

        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
```

```
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=rhh,
            selector=prm_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [58]:

```
# Initialize the results list
results_gsgsp_rhh = []

# Perform K-Fold cross-validation with the genetic programming algorithm
run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_rhh.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 25
)

results_gsgsp_rhh.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
```

```
gc.collect()

run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)

results_gsgsp_rhh.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_rhh = list(itertools.chain.from_iterable(results_gsgsp_rhh))

print("Cross-validation scores for all runs: ", results_gsgsp_rhh)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_rhh))
```

```
Cross-validation scores for run 1: [0.18756692 0.19670177 0.19185461 0.1934478  0.18
70664  0.18906228
 0.18800284 0.18528643 0.19100669 0.18984376]
Mean cross-validation score: 0.18998395
Cross-validation scores for run 2: [0.14402497 0.18829055 0.19276093 0.19071554 0.17
70847  0.19445647
 0.1728636  0.12256686 0.1944584  0.18858951]
Mean cross-validation score: 0.17658114
Cross-validation scores for run 3: [0.19176936 0.18693246 0.19110143 0.1889952  0.19
471295 0.18959932
 0.18926197 0.10959333 0.1190904  0.18489367]
Mean cross-validation score: 0.17459503
Cross-validation scores for all runs:  [0.18756692, 0.19670177, 0.19185461, 0.193447
8, 0.1870664, 0.18906228, 0.18800284, 0.18528643, 0.19100669, 0.18984376, 0.1440249
7, 0.18829055, 0.19276093, 0.19071554, 0.1770847, 0.19445647, 0.1728636, 0.12256686,
0.1944584, 0.18858951, 0.19176936, 0.18693246, 0.19110143, 0.1889952, 0.19471295, 0.
18959932, 0.18926197, 0.10959333, 0.1190904, 0.18489367]
Mean cross-validation score for all runs:  0.18038672
```

FULL intializations:

In [59]:

```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.4,
    'max_init_depth': 2,
```

```
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}

gsgp_params = {
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.2,
    'mutation_prob': 0.3,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}

#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,

    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic

        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
```

```
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=full,
            selector=prm_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [60]:

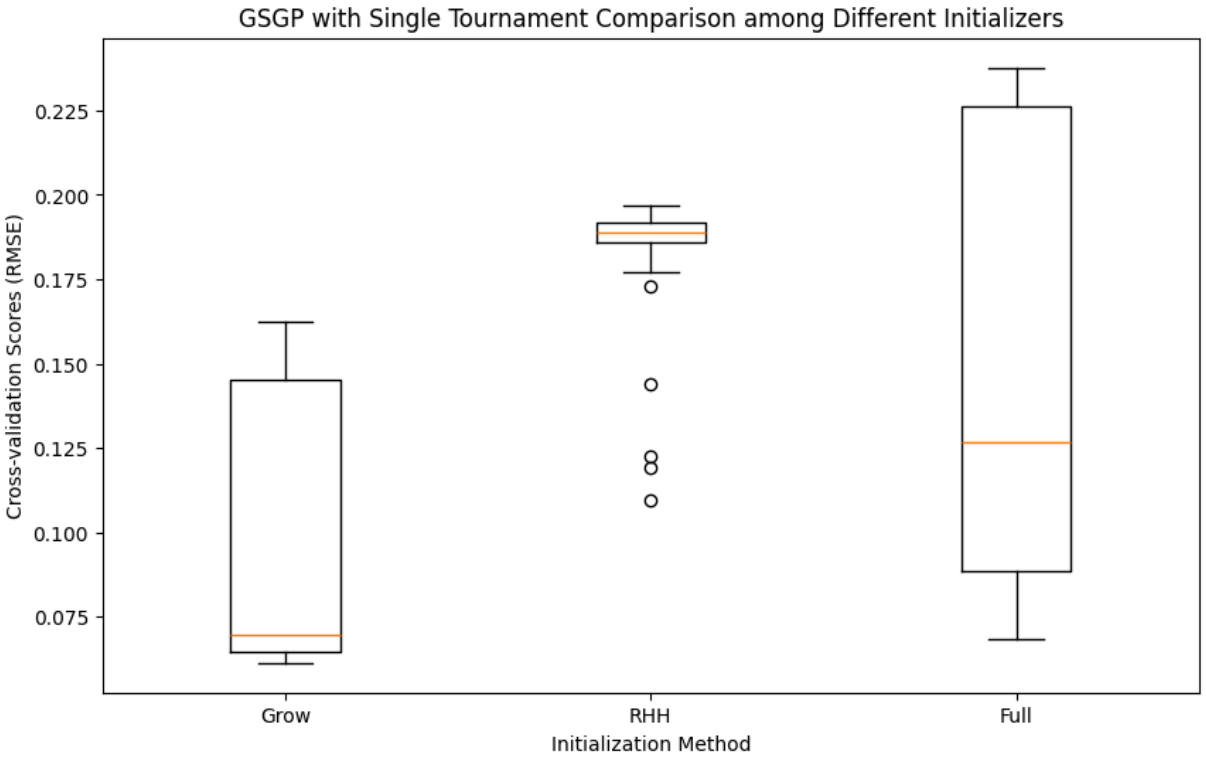
```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gsgsp_full = []

run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_full.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 25
```

```
)


results_gsgsp_full.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)

results_gsgsp_full.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_full = list(itertools.chain.from_iterable(results_gsgsp_full))

print("Cross-validation scores for all runs: ", results_gsgsp_full)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_full))
```

```
Cross-validation scores for run 1: [0.20213425 0.10834539 0.10945994 0.11405835 0.11
12104  0.23237652
 0.06847372 0.22290795 0.09081583 0.23766112]
Mean cross-validation score: 0.14974435
Cross-validation scores for run 2: [0.22747755 0.06934604 0.18465094 0.07241219 0.07
18564  0.07287286
 0.21740961 0.23687483 0.23120914 0.22904445]
Mean cross-validation score: 0.16131541
Cross-validation scores for run 3: [0.17768899 0.10894733 0.22742963 0.07135418 0.14
208475 0.07043084
 0.0875114  0.22878712 0.13964556 0.0934425 ]
Mean cross-validation score: 0.13473222
Cross-validation scores for all runs:  [0.20213425, 0.10834539, 0.109459944, 0.11405
835, 0.1112104, 0.23237652, 0.06847372, 0.22290795, 0.090815835, 0.23766112, 0.22747
755, 0.06934604, 0.18465094, 0.07241219, 0.0718564, 0.07287286, 0.21740961, 0.236874
83, 0.23120914, 0.22904445, 0.17768899, 0.10894733, 0.22742963, 0.07135418, 0.142084
75, 0.07043084, 0.0875114, 0.22878712, 0.13964556, 0.0934425]
Mean cross-validation score for all runs:  0.14859731
```

In [61]:
```
# Combine the scores into a list
data = [results_gsgsp_grow_optimized, results_gsgsp_rhh, results_gsgsp_full]

# Labels for each method
labels = ['Grow', 'RHH', 'Full']
```

```
# Create the boxplot
plt.figure(figsize=(10, 6))
plt.boxplot(data, labels=labels)
plt.title('GSGP with Single Tournament Comparison among Different Initializers')
plt.ylabel('Cross-validation Scores (RMSE)')
plt.xlabel('Initialization Method')
plt.grid(False)
plt.show()
```



GSGP Double Tournament Selection

Grow Initializations

In [62]:
```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.4,
    'max_init_depth': 2,
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}
```

```
gsgp_params = {
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.2,
    'mutation_prob': 0.3,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}

#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,

    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic

        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
```

```
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=grow,
            selector=prm_double_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [63]:

```
# Initialize the results list
results_gsgsp_double_grow = []

# Perform K-Fold cross-validation with the genetic programming algorithm
run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_double_grow.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

# Perform K-Fold cross-validation with the genetic programming algorithm
run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 25
)

results_gsgsp_double_grow.append(run_2)

print("Cross-validation scores for run 2:", run_2)
```

```
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

# Perform K-Fold cross-validation with the genetic programming algorithm
run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)

results_gsgsp_double_grow.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_double_grow = list(itertools.chain.from_iterable(results_gsgsp_double

print("Cross-validation scores for all runs: ", results_gsgsp_double_grow)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_double_gr
```

```
Cross-validation scores for run 1: [0.1665833  0.16995665 0.16528349 0.1519766  0.17
051505 0.14007519
 0.16340932 0.13699108 0.16520037 0.13589802]
Mean cross-validation score: 0.1565889
Cross-validation scores for run 2: [0.14197862 0.15630381 0.14685008 0.15356918 0.15
780571 0.16508584
 0.14230825 0.13205437 0.1415312  0.14303085]
Mean cross-validation score: 0.1480518
Cross-validation scores for run 3: [0.14381973 0.15635698 0.13556966 0.1616084  0.16
696039 0.16234191
 0.15456687 0.14152247 0.1402064  0.1643032 ]
Mean cross-validation score: 0.15272559
Cross-validation scores for all runs:  [0.1665833, 0.16995665, 0.16528349, 0.151976
6, 0.17051505, 0.14007519, 0.16340932, 0.13699108, 0.16520037, 0.13589802, 0.1419786
2, 0.15630381, 0.14685008, 0.15356918, 0.15780571, 0.16508584, 0.14230825, 0.1320543
7, 0.1415312, 0.14303085, 0.14381973, 0.15635698, 0.13556966, 0.1616084, 0.16696039,
0.16234191, 0.15456687, 0.14152247, 0.1402064, 0.1643032]
Mean cross-validation score for all runs:  0.15245542
```

RHH initializations:

In [64]:

```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
```

```
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.4,
    'max_init_depth': 2,
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}

gsgp_params = {
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.2,
    'mutation_prob': 0.3,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}

#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,

    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic

        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
```

```
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=rhh,
            selector=prm_double_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [65]:

```
# Initialize the results list
results_gsgsp_double_rhh = []

# Perform K-Fold cross-validation with the genetic programming algorithm
run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_double_rhh.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))

del run_1
gc.collect()

run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
```

```
    shuffle = True,
    random_state = 25
)

results_gsgsp_double_rhh.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)

results_gsgsp_double_rhh.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_double_rhh = list(itertools.chain.from_iterable(results_gsgsp_double_

print("Cross-validation scores for all runs: ", results_gsgsp_double_rhh)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_double_rh
```

```
Cross-validation scores for run 1: [0.2300733  0.2427795  0.24842215 0.20443209 0.24
322113 0.15472329
 0.25227845 0.17882414 0.24926059 0.17226098]
Mean cross-validation score: 0.21762756
Cross-validation scores for run 2: [0.3063339  0.24486534 0.18504779 0.24702537 0.22
736734 0.24355586
 0.23465878 0.25278616 0.3009377  0.31628838]
Mean cross-validation score: 0.25588667
Cross-validation scores for run 3: [0.1621192  0.25040793 0.31162333 0.2473164  0.27
200833 0.19705687
 0.21260698 0.2962608  0.3641108  0.24800949]
Mean cross-validation score: 0.256152
Cross-validation scores for all runs:  [0.2300733, 0.2427795, 0.24842215, 0.2044320
9, 0.24322113, 0.15472329, 0.25227845, 0.17882414, 0.24926059, 0.17226098, 0.306333
9, 0.24486534, 0.18504779, 0.24702537, 0.22736734, 0.24355586, 0.23465878, 0.2527861
6, 0.3009377, 0.31628838, 0.1621192, 0.25040793, 0.31162333, 0.2473164, 0.27200833,
0.19705687, 0.21260698, 0.2962608, 0.3641108, 0.24800949]
Mean cross-validation score for all runs:  0.24322207
```

FULL intializations:

In [66]:
```
# Convert the datasets to numpy arrays
X = train.values
```

```
y = y_lactose.values.flatten()

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml_gs = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.4,
    'max_init_depth': 2,
    'max_depth': 10,
    'n_batches': 1,
    'device': device
}

gsgp_params = {
    'sspace': sspace_sml_gs,
    'selection_pressure': 0.2,
    'mutation_prob': 0.3,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': device,
    'seed': 42
}

#Defining GSM steps:
to, by = 5.0, 0.25
ms = torch.arange(by, to + by, by, device=device)

def cross_val_gs_genetic_programming(X, y, cv=10, shuffle=True, random_state = 42,

    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        #initialize array where we will store scaled data
        X_scaled = np.empty_like(X)

        #split train and validation
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        #apply scaling on train and validation
        scaler = RobustScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_val_scaled = scaler.transform(X_val)

        #add scaled data to the initial array
        X_scaled[train_index] = X_train_scaled
        X_scaled[val_index] = X_val_scaled

        #save indices as a tensor
        train_indices = torch.tensor(train_index, dtype=torch.long).to(gp_kwargs['d
        val_indices = torch.tensor(val_index, dtype=torch.long).to(gp_kwargs['devic
```

```
        #save data as a tensor
        X_tensor = torch.tensor(X_scaled, dtype=torch.float32).to(gp_kwargs['device
        y_tensor = torch.tensor(y, dtype=torch.float32).to(gp_kwargs['device'])

        # Initialize genetic programming with the current fold's data
        pi_sml_gs = SMLGS(
            sspace=sspace_sml_gs,
            ffunction=Ffunctions('rmse'),
            X=X_tensor, y=y_tensor,
            train_indices=train_indices,
            test_indices=val_indices)

        mheuristic = GSGP(
            pi=pi_sml_gs,
            #add if we want to reconsctruct:
            #path_init_pop=path_init_pop,
            #path_rts=path_rts,
            initializer=full,
            selector=prm_double_tournament(gp_kwargs['selection_pressure']),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed'],
            crossover=prm_efficient_gs_xo(X_tensor, prm_grow(sspace=pi_sml_gs.sspac
            mutator=prm_efficient_gs_mtn(X_tensor, prm_grow(sspace=pi_sml_gs.sspace
)

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array([t.detach().cpu().numpy() for t in scores])
```

In [67]:

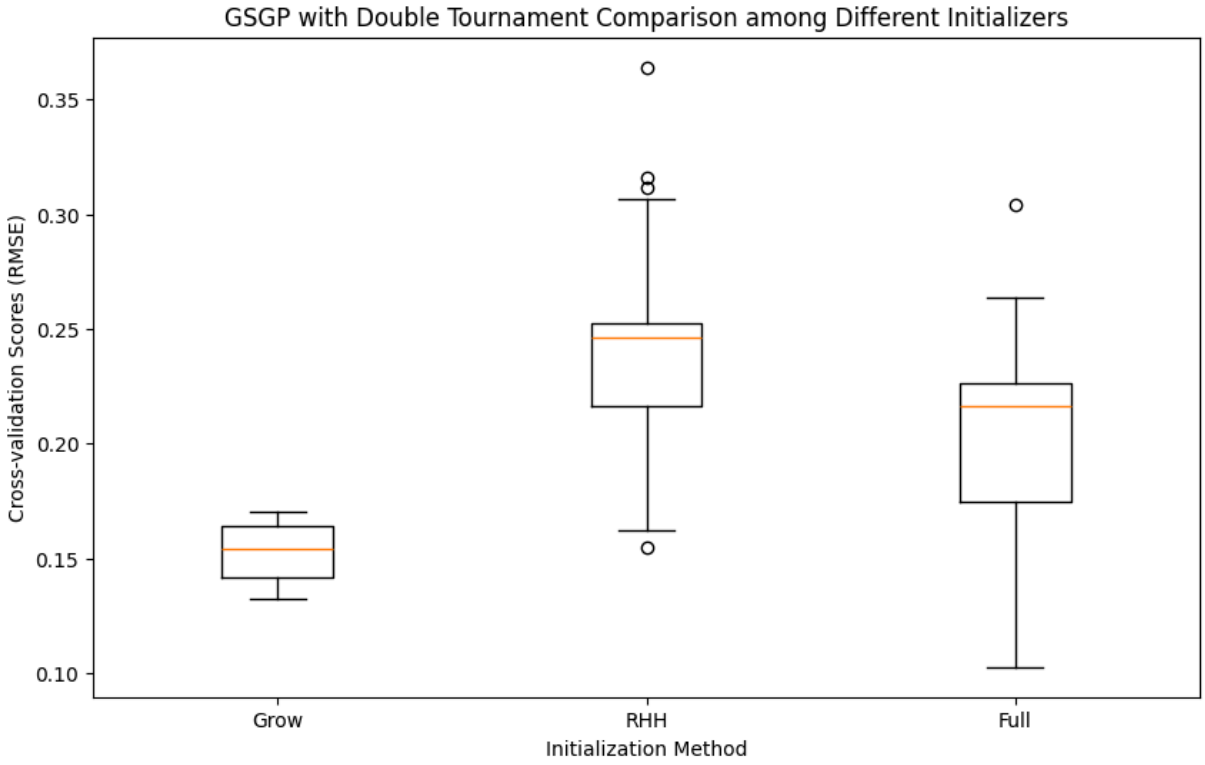
```
# Perform K-Fold cross-validation with the genetic programming algorithm
# Perform 3 times with shuffle=True and different random states to have 30 trials a

# Initialize the results list
results_gsgsp_double_full = []

run_1 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 42
)

results_gsgsp_double_full.append(run_1)

print("Cross-validation scores for run 1:", run_1)
print("Mean cross-validation score:", np.mean(run_1))
```

```
del run_1
gc.collect()

run_2 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 25
)


results_gsgsp_double_full.append(run_2)

print("Cross-validation scores for run 2:", run_2)
print("Mean cross-validation score:", np.mean(run_2))

del run_2
gc.collect()

run_3 = cross_val_gs_genetic_programming(
    X=X,
    y=y,
    **gsgp_params,
    shuffle = True,
    random_state = 63
)

results_gsgsp_double_full.append(run_3)

print("Cross-validation scores for run 3:", run_3)
print("Mean cross-validation score:", np.mean(run_3))

del run_3
gc.collect()

results_gsgsp_double_full = list(itertools.chain.from_iterable(results_gsgsp_double

print("Cross-validation scores for all runs: ", results_gsgsp_double_full)
print("Mean cross-validation score for all runs: ", np.mean(results_gsgsp_double_fu
```

```
Cross-validation scores for run 1: [0.1327279  0.22187622 0.1732881  0.13138744 0.22
600493 0.25909874
 0.19107513 0.21773526 0.23834652 0.21138908]
Mean cross-validation score: 0.20029294
Cross-validation scores for run 2: [0.1892075  0.23964691 0.11584513 0.21954966 0.26
338995 0.24187307
 0.10253242 0.21538013 0.17643547 0.24764633]
Mean cross-validation score: 0.20115066
Cross-validation scores for run 3: [0.19803765 0.22060537 0.1694205  0.17415637 0.13
565016 0.2261272
 0.19544493 0.22214626 0.30413848 0.22271048]
Mean cross-validation score: 0.20684373
Cross-validation scores for all runs:  [0.1327279, 0.22187622, 0.1732881, 0.1313874
4, 0.22600493, 0.25909874, 0.19107513, 0.21773526, 0.23834652, 0.21138908, 0.189207
5, 0.23964691, 0.11584513, 0.21954966, 0.26338995, 0.24187307, 0.102532424, 0.215380
13, 0.17643547, 0.24764633, 0.19803765, 0.22060537, 0.1694205, 0.17415637, 0.1356501
6, 0.2261272, 0.19544493, 0.22214626, 0.30413848, 0.22271048]
Mean cross-validation score for all runs:  0.20276245
```

In [68]:

```python
# Combine the scores into a list
data = [results_gsgsp_double_grow, results_gsgsp_double_rhh, results_gsgsp_double_f

# Labels for each method
labels = ['Grow', 'RHH', 'Full']

# Create the boxplot
plt.figure(figsize=(10, 6))
plt.boxplot(data, labels=labels)
plt.title('GSGP with Double Tournament Comparison among Different Initializers')
plt.ylabel('Cross-validation Scores (RMSE)')
plt.xlabel('Initialization Method')
plt.grid(False)
plt.show()
```



5. Neural Networks

Using PyTorch

In [69]:
```
#Prepare the data
seed = 42
#device = 'cuda' if torch.cuda.is_available() else 'cpu'
device = 'mps' if torch.backends.mps.is_available() else 'cpu' #para mac
#total_batches = 1

# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values.flatten()

# Convert to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)

dataset = TensorDataset(X_tensor, y_tensor)
```

In [70]:
```
#Define the Neural Network: simple feed-foward NN with one hidden layer
class RegressionNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        #Instanciate the parent (nn.Module) class
        super(RegressionNN, self).__init__()
        #1) NN architeture definition
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
```

```
        self.fc2 = nn.Linear(hidden_dim, output_dim)

        #self.apply(self._init_weights) #apply the function in 2)

    #2) Weights and bias initialization:
    def _init_weights(self, attribute):
        if isinstance(attribute, nn.Linear):
          torch.nn.init.xavier_uniform_(attribute.weight)#the variance of the weigh
          torch.nn.init.zeros_(attribute.bias) #bias are initialized as zero

    #3) Foward pass
    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

Working with functions that take into consideration Loss and RMSE calculations:

In [71]:
```
#4) Training Fucntion

#w/ RMSE
def train_model(model, criterion, optimizer, dataloader):
    model.train()
    total_loss = 0
    total_rmse = 0
    for inputs, targets in dataloader:
        targets = targets.view(-1, 1)  # Reshape targets to match model output
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        # Calculate RMSE
        rmse = torch.sqrt(F.mse_loss(outputs, targets))
        total_rmse += rmse.item()
    avg_loss = total_loss / len(dataloader)
    avg_rmse = total_rmse / len(dataloader)
    return avg_loss, avg_rmse

#5) Evaluation Function

#w/ RMSE
def evaluate_model(model, criterion, dataloader):
    model.eval()
    total_loss = 0
    total_rmse = 0
    with torch.no_grad():
        for inputs, targets in dataloader:
            targets = targets.view(-1, 1)  # Reshape targets to match model output
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            total_loss += loss.item()
```

```
            # Calculate RMSE
            rmse = torch.sqrt(F.mse_loss(outputs, targets))
            total_rmse += rmse.item()
    avg_loss = total_loss / len(dataloader)
    avg_rmse = total_rmse / len(dataloader)
    return avg_loss, avg_rmse
```

In [72]:

```
def cross_validate(dataset, k_folds=10, epochs=500, batch_size=16, learning_rate=0.
    kf = KFold(n_splits=k_folds, shuffle=True, random_state=random_state)
    fold_results = {}

    nn_names = ['SGD', 'MiniSGD', 'ASGD', 'RMSprop']
    for nn_name in nn_names:
        fold_results[nn_name] = {'losses': [], 'median_losses': [], 'train_rmse': [

    for fold, (train_idx, val_idx) in enumerate(kf.split(dataset)):
        train_subset = torch.utils.data.Subset(dataset, train_idx)
        val_subset = torch.utils.data.Subset(dataset, val_idx)

        train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True
        val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)

        input_dim = dataset.tensors[0].shape[1]
        hidden_dim = 20
        output_dim = 1

        models = {k: RegressionNN(input_dim, hidden_dim, output_dim) for k in nn_na
        criterion = nn.MSELoss()
        optimizers = {
            'SGD': torch.optim.SGD(models['SGD'].parameters(), lr=learning_rate),
            'MiniSGD': torch.optim.SGD(models['MiniSGD'].parameters(), lr=learning_
            'ASGD': torch.optim.ASGD(models['ASGD'].parameters(), lr=learning_rate)
            'RMSprop': torch.optim.RMSprop(models['RMSprop'].parameters(), lr=learn
        }

        for nn_name in fold_results.keys():
            models[nn_name].apply(models[nn_name]._init_weights)  # Ensure weights
            epoch_losses = []
            train_rmse = []
            validation_rmse=[]
            for epoch in range(epochs):
                train_loss, rmse = train_model(models[nn_name], criterion, optimize
                val_loss, val_rmse = evaluate_model(models[nn_name], criterion, val
                epoch_losses.append(val_loss)
                train_rmse.append(rmse)
                validation_rmse.append(val_rmse)

            fold_results[nn_name]['train_rmse'].append(train_rmse)
            fold_results[nn_name]['val_rmse'].append(validation_rmse)
            fold_results[nn_name]['losses'].append(epoch_losses)

    for nn_name, results in fold_results.items():
        median_losses = np.median(np.array(results['losses']), axis=0)
        fold_results[nn_name]['median_losses'] = median_losses
        median_rmse_val = np.median(np.array(results['val_rmse']), axis=0)
        fold_results[nn_name]['median_rmse_val'] = median_rmse_val
```

```
        median_rmse_train = np.median(np.array(results['train_rmse']), axis=0)
        fold_results[nn_name]['median_rmse_train'] = median_rmse_train
    return fold_results
```

In [73]:
```
# Execute cross-validation. At this stage we're figuring out which of the optimizer
results_2 = cross_validate(dataset)
```

In [74]:
```
minisgd=[]
asgd=[]
rmsprop=[]
sgd=[]


for i in range(10):
    minisgd.append(results_2.get('MiniSGD').get('train_rmse')[i][-1])
    asgd.append(results_2.get('ASGD').get('train_rmse')[i][-1])
    rmsprop.append(results_2.get('RMSprop').get('train_rmse')[i][-1])
    sgd.append(results_2.get('SGD').get('train_rmse')[i][-1])
```

In [75]:
```
np.mean(minisgd), np.mean(asgd), np.mean(rmsprop), np.mean(sgd)
```

Out[75]:
```
(0.22856130417436363,
 0.21678329724818468,
 1.0536395159363745,
 0.15184645131230354)
```

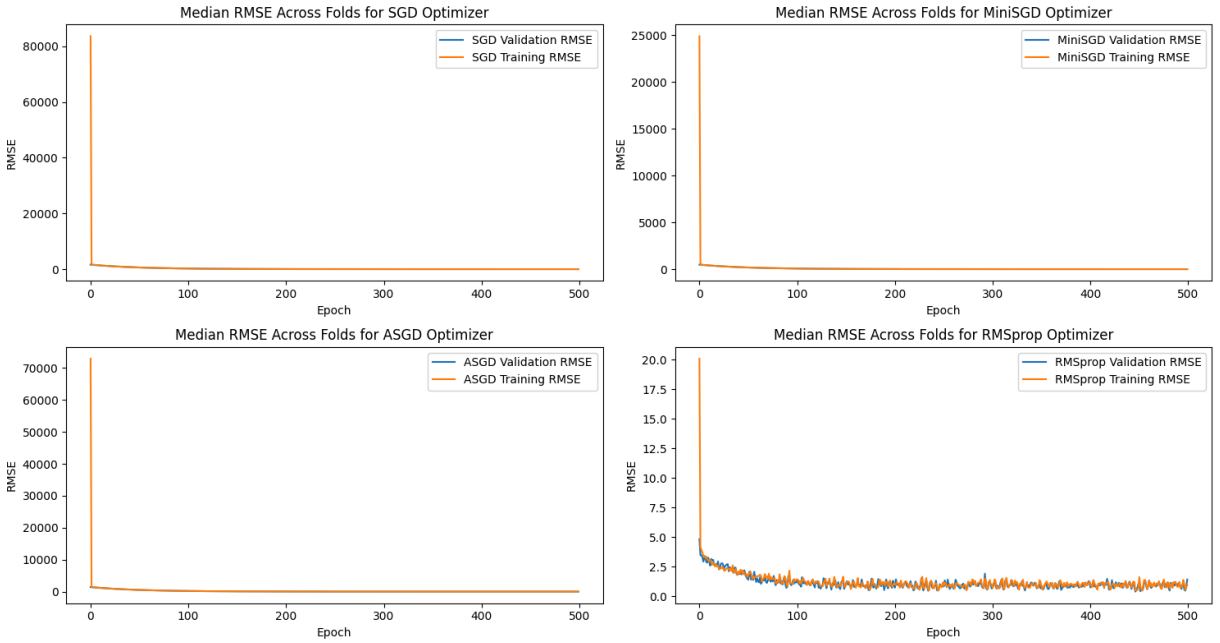
In [76]:
```
min(np.mean(minisgd), np.mean(asgd), np.mean(rmsprop), np.mean(sgd))
```

Out[76]:
```
0.15184645131230354
```

Best is SGD according to `rmse`. We will use that as our final NN model.

In [77]:
```
#LEARNING RATE : 0.000001, 500 epochs
nn_names = list(results_2.keys())
num_plots = len(nn_names)
num_cols = 2
num_rows = (num_plots + 1) // num_cols

fig, axs = plt.subplots(num_rows, num_cols, figsize=(15, 8))

for i, nn_name in enumerate(nn_names):
    row = i // num_cols
    col = i % num_cols
    if num_rows > 1:
        ax = axs[row, col]
    else:
        ax = axs[col]

    median_rmse_val = results_2[nn_name]['median_rmse_val']
    median_rmse_train = results_2[nn_name]['median_rmse_train']

    ax.plot(median_rmse_val, label=f'{nn_name} Validation RMSE')
    ax.plot(median_rmse_train, label=f'{nn_name} Training RMSE')

    ax.set_xlabel('Epoch')
```

```
    ax.set_ylabel('RMSE')
    ax.legend()
    ax.set_title(f'Median RMSE Across Folds for {nn_name} Optimizer')

plt.tight_layout()
plt.show()
```



They basically all converge straight away....

Running NN SGD 30 trials

In [78]:
```
results_nn = []

a = 0
b = 0
c = 0

run_1 = cross_validate(dataset, random_state = 42)

for i in range(10):

    a = run_1.get('SGD').get('train_rmse')[i][-1]
    results_nn.append(a)

print("Cross-validation scores for run 1:", results_nn)
print("Mean cross-validation score:", np.mean(results_nn))


del run_1
gc.collect()

run_2 = cross_validate(dataset, random_state = 42)

for i in range(10):
```

```
    b = run_2.get('SGD').get('train_rmse')[i][-1]
    results_nn.append(b)

print("Cross-validation scores for run 2:", results_nn[10:20])
print("Mean cross-validation score for run 2:", np.mean(results_nn[10:20]))


del run_2
gc.collect()


run_3 = cross_validate(dataset, random_state = 42)

for i in range(10):

    c = run_3.get('SGD').get('train_rmse')[i][-1]
    results_nn.append(c)

print("Cross-validation scores for run 1:", results_nn[21:30])
print("Mean cross-validation score:", np.mean(results_nn[21:30]))


del run_3
gc.collect()


print("Cross-validation scores for all runs: ", results_nn)
print("Mean cross-validation score for all runs: ", np.mean(results_nn))
```

```
Cross-validation scores for run 1: [0.20871215909719468, 0.06434727273881435, 0.0657
1685746312142, 0.0629822075366974, 0.3496130704879761, 0.38442668318748474, 0.108054
26761507989, 0.0812776580452919, 0.08773290850222111, 0.06403698697686196]
Mean cross-validation score: 0.14769000716507436
Cross-validation scores for run 2: [0.11349569633603096, 0.9814033448696137, 0.07345
01414000988, 0.1780591666698456, 0.10558877289295196, 0.36284488439559937, 2.1901139
73617554, 4.17493782043457, 1.5907501101493835, 0.18202198445796966]
Mean cross-validation score for run 2: 0.9952665895223618
Cross-validation scores for run 1: [0.06466595903038978, 0.16750582456588745, 0.2809
076517820358, 0.07564040645956993, 0.06135126799345016, 0.10449615269899368, 0.07956
90432190895, 0.06362922675907612, 0.4776715785264969]
Mean cross-validation score: 0.1528263456705544
Cross-validation scores for all runs:  [0.20871215909719468, 0.06434727273881435, 0.
06571685746312142, 0.0629822075366974, 0.3496130704879761, 0.38442668318748474, 0.10
805426761507989, 0.0812776580452919, 0.08773290850222111, 0.06403698697686196, 0.113
49569633603096, 0.9814033448696137, 0.0734501414000988, 0.1780591666698456, 0.105588
77289295196, 0.36284488439559937, 2.190113973617554, 4.17493782043457, 1.59075011014
93835, 0.18202198445796966, 0.06272088028490544, 0.06466595903038978, 0.167505824565
88745, 0.2809076517820358, 0.07564040645956993, 0.06135126799345016, 0.1044961526989
9368, 0.0795690432190895, 0.06362922675907612, 0.4776715785264969]
Mean cross-validation score for all runs:  0.42892413193980855
```

6. NEAT

Running NEAT with train_test split. We were not able to perform NEAT using K-Folds.

In [79]:
```
train = pd.read_csv("train_preprocessed.csv")
```

In [80]:
```
X_train, X_test, y_train, y_test = train_test_split(train, y_lactose, test_size=0.3
```

In [81]:
```
print(f'Train shape {X_train.shape}')
print(f'Test shape {X_test.shape}')

print(f'Train target shape {y_train.shape}')
print(f'Test target shape {y_test.shape}')
```

```
Train shape (123, 19)
Test shape (54, 19)
Train target shape (123, 1)
Test target shape (54, 1)
```

In [82]:
```
Scaler = RobustScaler()
```

In [83]:
```
# Identify the non one-hot encoded features (continuous features)
continuous_features = ['delivery_age_years', 'dry_days', 'forage_kg_day', 'ruminati
                       'milk_kg_day', 'milk_kg_min_robot', 'milkings_day', 'errors_
                       'high_cdt_by_100_milkings', 'watery_by_100_milkings', 'refus

# Apply RobustScaler to the continuous features
scaler = RobustScaler()

# Fit the scaler on the training data and transform the train, validation, and test
X_train_continuous = scaler.fit_transform(X_train[continuous_features])
X_test_continuous = scaler.transform(X_test[continuous_features])

# Replace the continuous features in the original datasets with the scaled values
X_train[continuous_features] = X_train_continuous
X_test[continuous_features] = X_test_continuous
```

In [84]:
```
# Verify the transformation
print("Transformed training set:")
X_train.head()
```

```
Transformed training set:
```

Out[84]:

	delivery_age_years	dry_days	forage_kg_day	rumination_min_day	milk_kg_day	milk_
128	-0.177778	0.000000	-0.605490	-2.515543	-1.315585	
104	-0.311111	0.526316	0.395571	-0.026134	0.400223	
78	0.311111	-0.105263	0.675962	-0.055603	0.158097	
36	0.311111	-0.421053	1.293534	-0.849739	0.608584	
93	0.311111	0.842105	-0.248765	0.306088	0.467258	

In [85]:
```
print("Transformed test set:")
X_test.head()
```

```
Transformed test set:
```

Out[85]:

	delivery_age_years	dry_days	forage_kg_day	rumination_min_day	milk_kg_day	milk_l
19	0.533333	-4.631579	-0.439382	-2.046404	-1.392813	
45	0.444444	0.842105	0.044291	0.151622	-0.336758	
139	-0.133333	-0.421053	-0.318766	-0.009403	0.189451	
30	1.333333	0.315789	-0.852656	0.425982	0.061661	
67	-0.177778	2.421053	0.688134	-0.530314	-0.206059	

In [86]:
```
X_train = X_train.values
X_test = X_test.values

y_train = y_train.values
y_test = y_test.values
```

In [87]:
```
config_file = 'config-feedforward-xor'
```

In [88]:
```
# Open the config file with the correct encoding
with open(config_file, encoding='utf-8') as f:
    content = f.read()
```

In [89]:
```
config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                     neat.DefaultSpeciesSet, neat.DefaultStagnation,
                     config_file)
```

In [90]:
```
def eval_rmse(net, X, y):
    '''
    Auxiliary function to evaluate the RMSE.
    '''
    fit = 0.
    for xi, xo in zip(X, y):
        output = net.activate(xi)
        fit += (output[0] - xo)**2

    # RMSE
    return (fit / len(y))**0.5


def eval_genomes(genomes, config):
    '''
    The function used by NEAT-Python to evaluate the fitness of the genomes.
    -> It has to have the two first arguments equals to the genomes and config obje
    -> It has to update the `fitness` attribute of the genome.
    '''
    for genome_id, genome in genomes:

        # Define the network
```

```
        net = neat.nn.FeedForwardNetwork.create(genome, config)

        # Train fitness
        train_rmse = eval_rmse(net, X_train, y_train)

        #Test fitness
        test_rmse = eval_rmse(net, X_test, y_test)

        # Ensure fitness is a scalar value
        genome.fitness = float(-test_rmse)
```

In [91]:
```
# Create the population, which is the top-level object for a NEAT run.
pop = neat.Population(config)
```

Running NEAT 30 times

In [92]:
```
# Add a stdout reporter to show progress in the terminal.
pop.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
pop.add_reporter(stats)
pop.add_reporter(neat.Checkpointer(200))
```

In [ ]:
```
results = []  # List to store the results of each iteration

for i in range(30):
    # Split the data with a new random state each time
    X_train, X_test, y_train, y_test = train_test_split(train, y_lactose, test_size

    # Define the continuous features
    continuous_features = ['delivery_age_years', 'dry_days', 'forage_kg_day', 'rumi
                           'milk_kg_day', 'milk_kg_min_robot', 'milkings_day', 'err
                           'high_cdt_by_100_milkings', 'watery_by_100_milkings', 'r

    # Apply RobustScaler to the continuous features
    scaler = RobustScaler()

    # Fit the scaler on the training data and transform the train, validation, and
    X_train_continuous = scaler.fit_transform(X_train[continuous_features])
    X_test_continuous = scaler.transform(X_test[continuous_features])

    # Replace the continuous features in the original datasets with the scaled valu
    X_train[continuous_features] = X_train_continuous
    X_test[continuous_features] = X_test_continuous

    # Convert dataframe to numpy array (if needed, assuming pandas DataFrames)
    X_train = X_train.values
    X_test = X_test.values
    y_train = y_train.values
    y_test = y_test.values

    winner = pop.run(eval_genomes, 100)

    result = winner.fitness
```

```
    results.append(result)
```

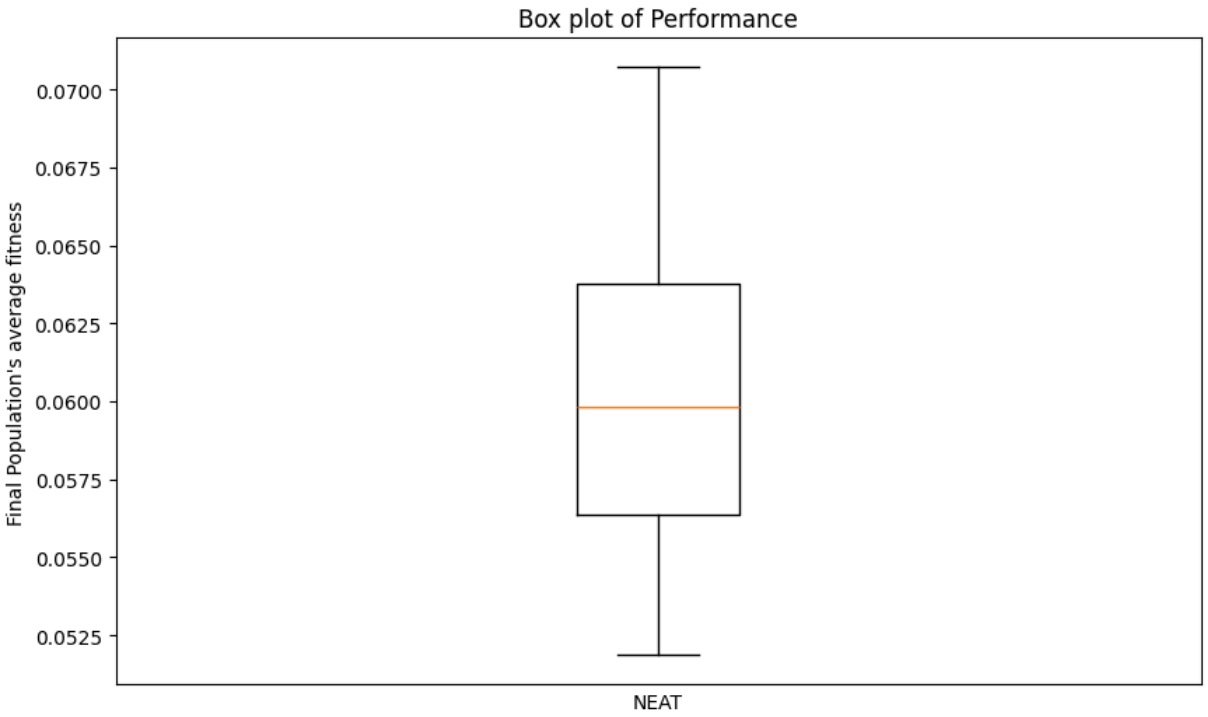
In [94]:
```
results_neat = np.abs(results)
```

In [95]:
```
len(results_neat), results_neat
```

Out[95]:
```
(30,
 array([0.06798898, 0.05532675, 0.05757827, 0.06964325, 0.06056779,
        0.06591813, 0.05817881, 0.06011596, 0.07070573, 0.06156239,
        0.06371916, 0.06378689, 0.0596313 , 0.06001495, 0.06223811,
        0.05459252, 0.06323153, 0.0640758 , 0.05434493, 0.0650843 ,
        0.05185872, 0.06415328, 0.05853024, 0.05624576, 0.05726532,
        0.05336995, 0.05671867, 0.05398014, 0.0559495 , 0.05869149]))
```

In [96]:
```
results_neat = results_neat.tolist()
```

In [97]:
```
# Create the boxplot
plt.figure(figsize=(10, 6))
plt.boxplot(results_neat)
plt.title('Box plot of Performance')
plt.ylabel("Final Population's average fitness")
plt.xlabel("NEAT")
plt.xticks([])
plt.grid(False)
plt.show()
```



7. Evaluation Metrics

In order to assess which of our algorithms fared better, we will perform a series of statistical tests on our result metric (RMSE), comparing all models.

Since we have so many models and different results, we will use the **Wilcoxon Signed-Rank test** to compare, in 1vs1 fashion, models where the only change is the max_init_depth (1vs1) or the initialization method. We will pick the best from there, and then apply the **Friedman test** to the "winners".

In [98]:
```
# All models results
all_results = {
    'GP Single Tournament, Grow, max_init_depth = 3': results_gp_single_tournament_
    'GP Single Tournament, Grow, max_init_depth = 5': results_gp_single_tournament_
    'GP Single Tournament, RHH': results_gp_single_tournament_rhh,
    'GP Single Tournament, Full': results_gp_single_tournament_full,
    'GP Double Tournament, Grow': results_gp_double_tournament_grow,
    'GP Double Tournament, RHH': results_gp_double_tournament_rhh,
    'GP Double Tournament, Full': results_gp_double_tournament_full,
    'GSGP Single Tournament, Grow' : results_gsgsp_grow,
    'GSGSP Single Tournament, Grow, optimized' : results_gsgsp_grow_optimized,
    'GSGSP Single Tournament, RHH' : results_gsgsp_rhh,
    'GSGP Single Tournament, Full' : results_gsgsp_full,
    'GSGP Double Tournament, Grow' : results_gsgsp_double_grow,
    'GSGP Double Tournament, RHH' : results_gsgsp_double_rhh,
    'GSGP Double Tournament, Full' : results_gsgsp_double_full,
    'NN' : results_nn,
    'NEAT' : results_neat
}
```

In [99]:
```
df_results = pd.DataFrame(all_results)
```

In [100…
```
df_results.to_csv('all_results.csv', index=False)
```

In [101…
```
df_results
```

Out[101...

	GP Single Tournament, Grow, max_init_depth = 3	GP Single Tournament, Grow, max_init_depth = 5	GP Single Tournament, RHH	GP Single Tournament, Full	GP Double Tournament, Grow	GP Double Tournament, RHH
0	0.000084	0.004858	0.007344	0.013595	0.084395	0.012567
1	0.005892	0.086029	0.116750	0.028424	0.099391	0.000650
2	0.000407	0.003804	0.004858	0.000129	0.084395	0.045638
3	0.000407	0.007776	0.052494	0.007826	0.084657	0.146809
4	0.000084	0.004858	0.004858	0.004858	0.086669	0.027203
5	0.000407	0.008059	0.023516	0.011368	0.099994	0.004858
6	0.000407	3.202504	0.004956	0.004858	0.084395	0.000882
7	0.009822	0.019143	0.007344	0.007344	0.086669	2.579684
8	0.000407	0.007344	0.010596	0.000674	0.084395	0.004858
9	0.000995	1.562469	0.006391	0.010596	0.085986	0.045338
10	0.000084	0.010495	0.007344	0.004858	0.007543	0.627474
11	0.000084	0.000979	0.004858	0.001942	0.089395	0.014721
12	0.000634	0.013212	0.004858	0.004858	0.084657	0.007826
13	0.000407	0.001859	0.021694	0.004858	0.086669	0.007344
14	0.000084	0.017443	0.004858	0.004858	0.086669	0.007344
15	0.003883	0.007344	0.018683	0.000397	0.084395	0.071969
16	0.000620	0.001038	0.011815	0.007344	0.086033	0.007344
17	0.000620	0.008603	0.069523	0.004858	0.099391	0.051239
18	0.000084	0.165430	0.003423	0.104937	0.001191	0.002535
19	0.000634	0.004858	0.003415	0.001180	0.086669	0.016872
20	0.000995	0.004858	0.001975	0.004858	0.084240	0.011633
21	0.000084	0.004858	0.004858	0.004858	0.118734	0.004858
22	0.000084	0.004858	0.004858	0.006040	2.594679	0.110760
23	0.001038	0.004858	0.007344	0.007344	0.085432	0.007344
24	0.000265	0.007344	0.017288	0.097190	0.084395	2.579684
25	0.005158	0.007344	0.017965	0.004858	0.099391	0.061388
26	0.002679	0.017953	0.011288	0.007826	0.086033	0.002502
27	0.005901	0.004858	0.010864	0.004858	0.085986	0.000615

	GP Single Tournament, Grow, max_init_depth = 3	GP Single Tournament, Grow, max_init_depth = 5	GP Single Tournament, RHH	GP Single Tournament, Full	GP Double Tournament, Grow	GP Double Tournament, RHH
28	0.000084	0.007344	0.008261	0.092241	0.085986	0.053169
29	0.001202	0.007344	0.004858	0.004858	0.087918	0.007826

Friedman Test to understand if there are statistically significant differences among the algorithm results

In [102…
```
friedmanchisquare(results_gp_single_tournament_grow_max_init_3, \
   results_gp_single_tournament_grow_max_init_5, results_gp_single_tournament_full,
       results_gsgsp_grow, results_gsgsp_grow_optimized, results_gsgsp_rhh, results
       results_gsgsp_double_grow, results_gsgsp_double_rhh, results_gsgsp_double_fu
        results_nn, results_neat)
```

Out[102…
```
FriedmanchisquareResult(statistic=245.8428171641792, pvalue=2.0722302215270414e-4
6)
```

Friedman Test shows, without a shadow of doubt (p-value<0.05, in fact very very small), that there are significant differences between algorithms.

Wilcoxon Signed-Rank Test Comparisons

GP Single Tournament, Grow, max_init_depth=3 vs **GP Single Tournament, Grow, max_init_depth=5**

In [103…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gp_single_tournament
```

Out[103…
```
WilcoxonResult(statistic=3.0, pvalue=9.313225746154785e-09)
```

This shows there are significant differences between the these algorithms. As we analyzed in our results above, we choose the one with `max_init_depth = 3`.

GP Single Tournament, Grow vs **GP Single Tournament, RHH**

In [104…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gp_single_tournament
```

Out[104…
```
WilcoxonResult(statistic=2.0, pvalue=5.587935447692871e-09)
```

Again, p-value <0.05 indicates statistically significant differences in outcomes. Looking at the mean across al runs, we stick with the `Grow` initializer.

GP Single Tournament, Grow vs **GP Single Tournament, Full**

In [105…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gp_single_tournament
```

Out[105… `WilcoxonResult(statistic=25.0, pvalue=1.6838312149047852e-06)`

Again, p-value <0.05 indicates statistically significant differences in outcomes. Looking at the mean across al runs, we stick with the `Grow` initializer.

GSGP, Grow vs **GSGP, Grow, Optimized**

In [106…
```
wilcoxon(results_gsgsp_grow, results_gsgsp_grow_optimized)
```

Out[106… `WilcoxonResult(statistic=19.0, pvalue=5.718320608139038e-07)`

Statistically significant differences. We stick with the **optimized** version.

GP Single Tournament, Grow vs **GSGP, Grow, Optimized**

In [107…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gsgsp_grow_optimized
```

Out[107… `WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)`

In [108…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gsgsp_grow_o
```

Out[108… `(0.0014512674, 0.10123125)`

Our GP Single Tournament Grow model is still to be beaten at this point.

GP Single Tournament, Grow vs **GSGP, RHH**

In [109…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gsgsp_rhh)
```

Out[109… `WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)`

In [110…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gsgsp_rhh)
```

Out[110… `(0.0014512674, 0.18038672)`

GP Single Tournament, Grow vs **GSGP, Full**

In [111…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gsgsp_full)
```

Out[111… `WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)`

In [112…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gsgsp_full)
```

Out[112… `(0.0014512674, 0.14859731)`

GP Single Tournament, Grow vs **GP, Double Tournament, Grow**

In [113…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gp_double_tournament
```

Out[113… `WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)`

In [114…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gp_double_to
```

Out[114…
```
(0.0014512674, 0.16687857)
```

GP Single Tournament, Grow vs **GP, Double Tournament, RHH**

In [115…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gp_double_tournament
```

Out[115…
```
WilcoxonResult(statistic=16.0, pvalue=3.1478703022003174e-07)
```

In [116…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gp_double_to
```

Out[116…
```
(0.0014512674, 0.21743117)
```

GP Single Tournament, Grow vs **GP, Double Tournament, Full**

In [117…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gp_double_tournament
```

Out[117…
```
WilcoxonResult(statistic=12.0, pvalue=1.30385160446167e-07)
```

In [118…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gp_double_to
```

Out[118…
```
(0.0014512674, 0.035840522)
```

GP Single Tournament, Grow vs **GSGP, Double Tournament, Grow**

In [119…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gsgsp_double_grow)
```

Out[119…
```
WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)
```

In [120…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gsgsp_double
```

Out[120…
```
(0.0014512674, 0.15245542)
```

GP Single Tournament, Grow vs **GSGP, Double Tournament, RHH**

In [121…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gsgsp_double_rhh)
```

Out[121…
```
WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)
```

In [122…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gsgsp_double
```

Out[122…
```
(0.0014512674, 0.24322207)
```

GP Single Tournament, Grow vs **GSGP, Double Tournament, Full**

In [123…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_gsgsp_double_full)
```

Out[123…
```
WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)
```

In [124…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_gsgsp_double
```

Out[124…
```
(0.0014512674, 0.20276245)
```

GP Single Tournament, Grow vs **Neural Network**

In [125…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_nn)
```

Out[125…
```
WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)
```

In [126…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_nn)
```

Out[126…
```
(0.0014512674, 0.42892413193980855)
```

GP Single Tournament, Grow vs **NEAT**

In [127…
```
wilcoxon(results_gp_single_tournament_grow_max_init_3, results_neat)
```

Out[127…
```
WilcoxonResult(statistic=0.0, pvalue=1.862645149230957e-09)
```

In [128…
```
np.mean(results_gp_single_tournament_grow_max_init_3), np.mean(results_neat)
```

Out[128…
```
(0.0014512674, 0.06016895398399467)
```

In light of these results, we conclude that the best performing algorithm is **Genetic Programming, Single Tournament Selection, Grow Initialization, with max_init_depth = 3**.

7. Best Model Plots

GP with single tournament selection, max_init_depth = 3

In [129…
```
# Convert the datasets to numpy arrays
X = train.values
y = y_lactose.values

# Genetic programming parameters
fset = [function_map['add'], function_map['sub'], function_map['mul'], function_map

sspace_sml = {
    'n_dims': train.shape[1],
    'function_set': fset, 'constant_set': ERC(-1., 1.),
    'p_constants': 0.1,
    'max_init_depth': 3,
    'max_depth': 10,
    'n_batches': 1,
    'device': 'cpu'
}
```

```
gp_params = {
    'sspace': sspace_sml,
    'selection_pressure': 0.07,
    'mutation_prob': 0.1,
    'xo_prob': 0.9,
    'has_elitism': True,
    'allow_reproduction': False,
    'pop_size': 141,
    'device': 'cpu',
    'seed': 42
}


def create_dataloaders(X_train, y_train, X_val, y_val, batch_size, shuffle=True):
    ds_train = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tens
    ds_val = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y
    dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=shuffle)
    dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=shuffle)
    return dl_train, dl_val

def cross_val_genetic_programming(X, y, cv=10, shuffle=True, random_state=42, **gp_
    kf = KFold(n_splits=cv, shuffle = True, random_state=random_state)
    scores = []

    for train_index, val_index in kf.split(X):
        X_train, X_val = X[train_index], X[val_index]
        y_train, y_val = y[train_index], y[val_index]

        scaler = RobustScaler()
        X_train = scaler.fit_transform(X_train)
        X_val = scaler.transform(X_val)

        # Ensure batch_size does not exceed the number of samples
        current_batch_size = 141

        dl_train, dl_val = create_dataloaders(X_train, y_train, X_val, y_val, curre

        # Initialize genetic programming with the current fold's data
        pi_sml = SML(
            sspace=gp_kwargs['sspace'],
            ffunction=Ffunctions('rmse'),
            dl_train=dl_train, dl_test=dl_val,
            n_jobs=8
        )

        mheuristic = GeneticAlgorithm(
            pi=pi_sml,
            initializer=grow,
            selector=prm_tournament(pressure=gp_kwargs['selection_pressure']),
            crossover=swap_xo,
            mutator=prm_subtree_mtn(initializer=prm_grow(gp_params['sspace'])),
            pop_size=gp_kwargs['pop_size'],
            p_m=gp_kwargs['mutation_prob'],
            p_c=gp_kwargs['xo_prob'],
            elitism=gp_kwargs['has_elitism'],
            reproduction=gp_kwargs['allow_reproduction'],
```

```
            device=gp_kwargs['device'],
            seed=gp_kwargs['seed']
        )

        # Solve the genetic programming algorithm for the current fold
        mheuristic.solve()
        fold_score = mheuristic.best_sol.fit
        scores.append(fold_score)

    return np.array(scores)
```

In [130…

```
# Retrieve the results on the validation after executing the best model with 10 fol
best_model_10_folds = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=10,
    shuffle=True,
    **gp_params,
    random_state=15
)
```

In [131…

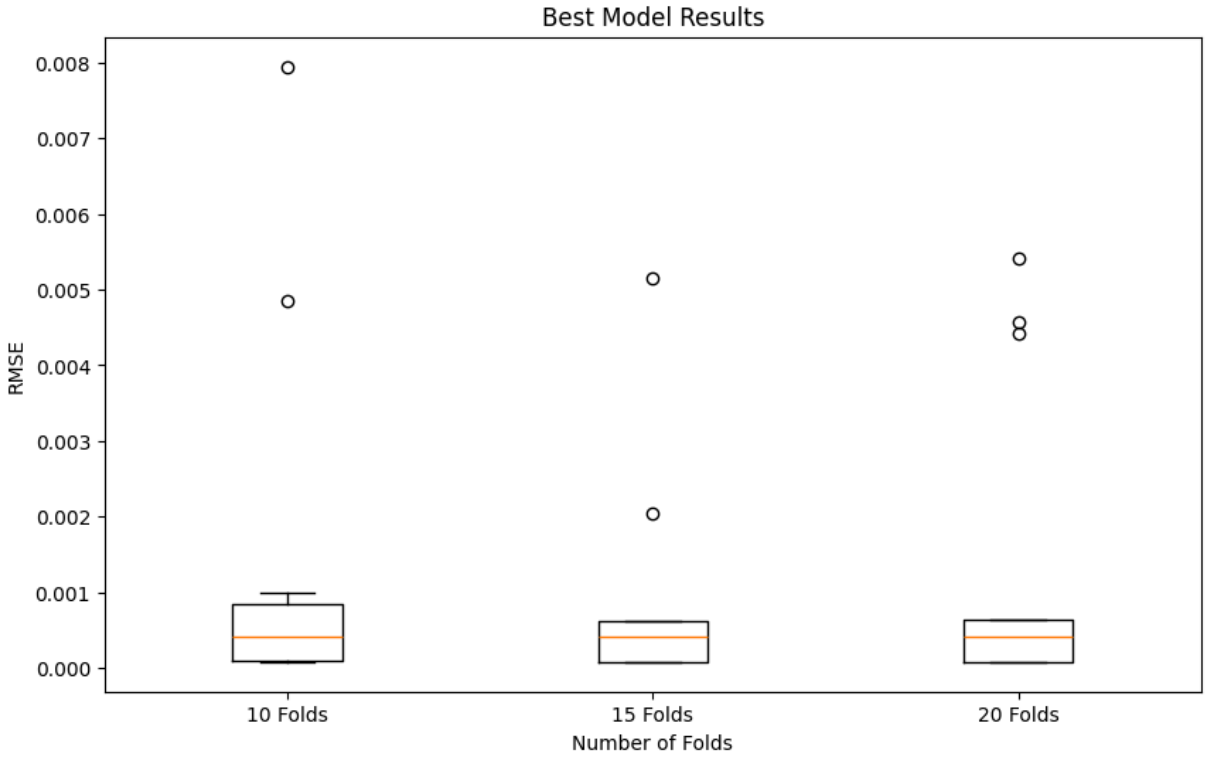
```
# Retrieve the results on the validation after executing the best model with 15 fol
best_model_15_folds = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=15,
    shuffle=True,
    **gp_params,
    random_state=15
)
```

In [132…

```
# Retrieve the results on the validation after executing the best model with 20 fol
best_model_20_folds = cross_val_genetic_programming(
    X=X,
    y=y,
    cv=20,
    shuffle=True,
    **gp_params,
    random_state=15
)
```

In [133…

```
# Combine execution results into a single list for plotting
results = [best_model_10_folds, best_model_15_folds, best_model_20_folds]

# Create a boxplot comparing the executions with different amount of folds
plt.figure(figsize=(10, 6))
plt.boxplot(results, labels=['10 Folds', '15 Folds', '20 Folds'])
plt.title('Best Model Results')
plt.xlabel('Number of Folds')
plt.ylabel('RMSE')
plt.grid(False)
plt.show()
```



In [134…

```
end_time = time.time()
total_time = end_time - start_time

print(f'Total time the notebook took to run: {round((total_time/60),2)} minutes.')
```

```
Total time the notebook took to run: 69.43 minutes.
```