Topics On Optimization and Machine Learning
Homework 4: Missing Value Imputation

Adrià Lisa

Introduction

For this assignment we are asked compare several methods that reconstruct missing values in time-series data. We mainly used `python` with the `pandas` and `scikit-learn` frameworks.

The dataset contains Ozone (O^3) concentration measurements for eight sensors, each one located at a different Barcelona district. We will use the provided python script to generate bursts of B missing values, with a certain probability, onto a new `missing` dataframe.

It is worth noting that in the following sections all the reported results are dependent on one instance of the `produce missings` function, with `perc_miss=0.1, length_miss = 1`, as some of the methods are very computationally intensive. In the final section, we will give a final comparison using different instances of `produce missings`.

Univariate Models

We will first consider three different methods that only work with the data of 1 sensor, interpreted as a time series with missing values. We will compare them in temrs of the usual MSE and R^2 test error metrics, which are computed by comparing the filled missing vaues with the original data. The results are summarized in figures 1 and 2.

Last Observed Carried Forward (LOCF)

This method is the simplest one, as it just sets every NaN value as the last non-NaN value in the time series. In case the first value is NaN, we assign to it the first non-NaN value in the time series (LOCB). With the `pandas` framework this is can be done quite easily:

```
locf = missing.copy().fillna(method='ffill')
locf.fillna(method='bfill', inplace=True)
```

As this method is the simplest, it will be considered as the baseline. All the other methods are more complicated, and if they do not improve on the results of this one, they should be discarded.

The average of the results were $MSE = 193.122$ and $R^2 = 0.719$.

Polynomial Interpolation

As instructed in the assignment, we considered the polynomial of degree 3 that serves as the regression curve of the time series. As the time-steps are evenly distributed (every hour), we can use the `interp1d` method with `kind='cubic'` quite easily to extrapolate on the missing values.

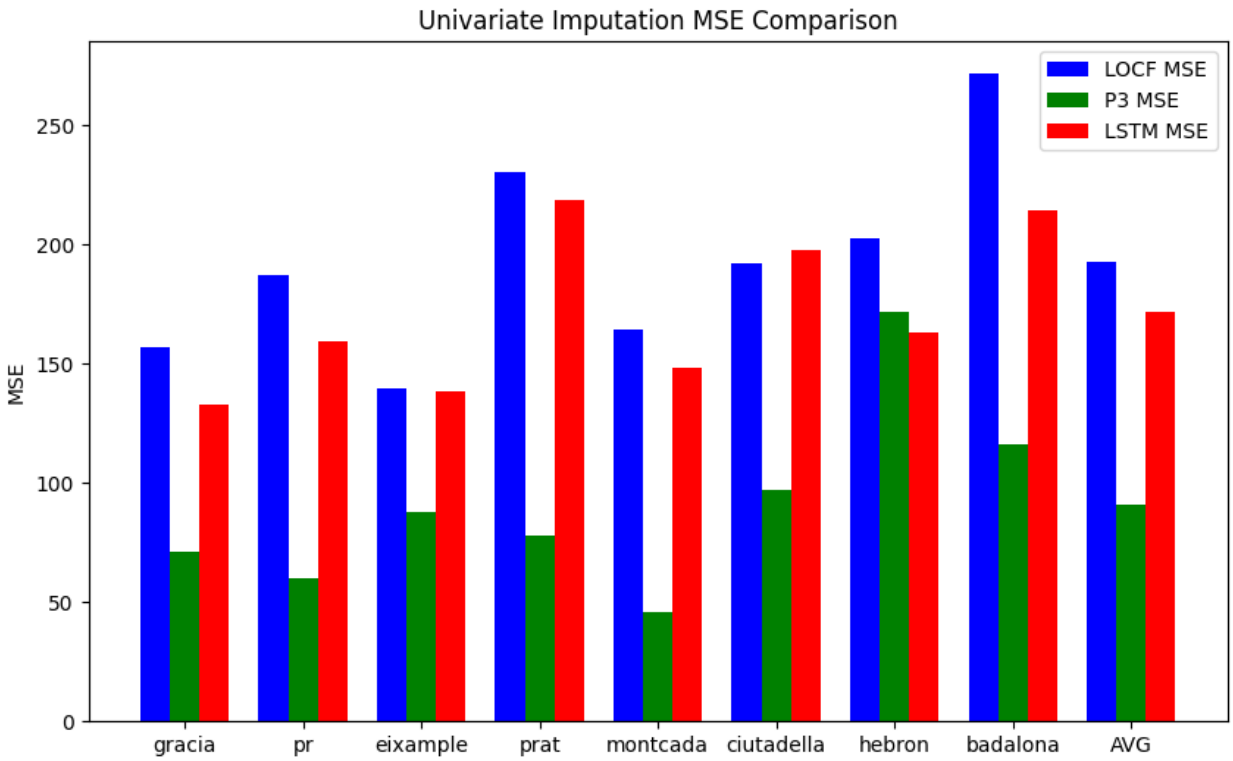

Figure 1: Mean Squared Error between the reconstructed missing values of 1 sample and the original values.

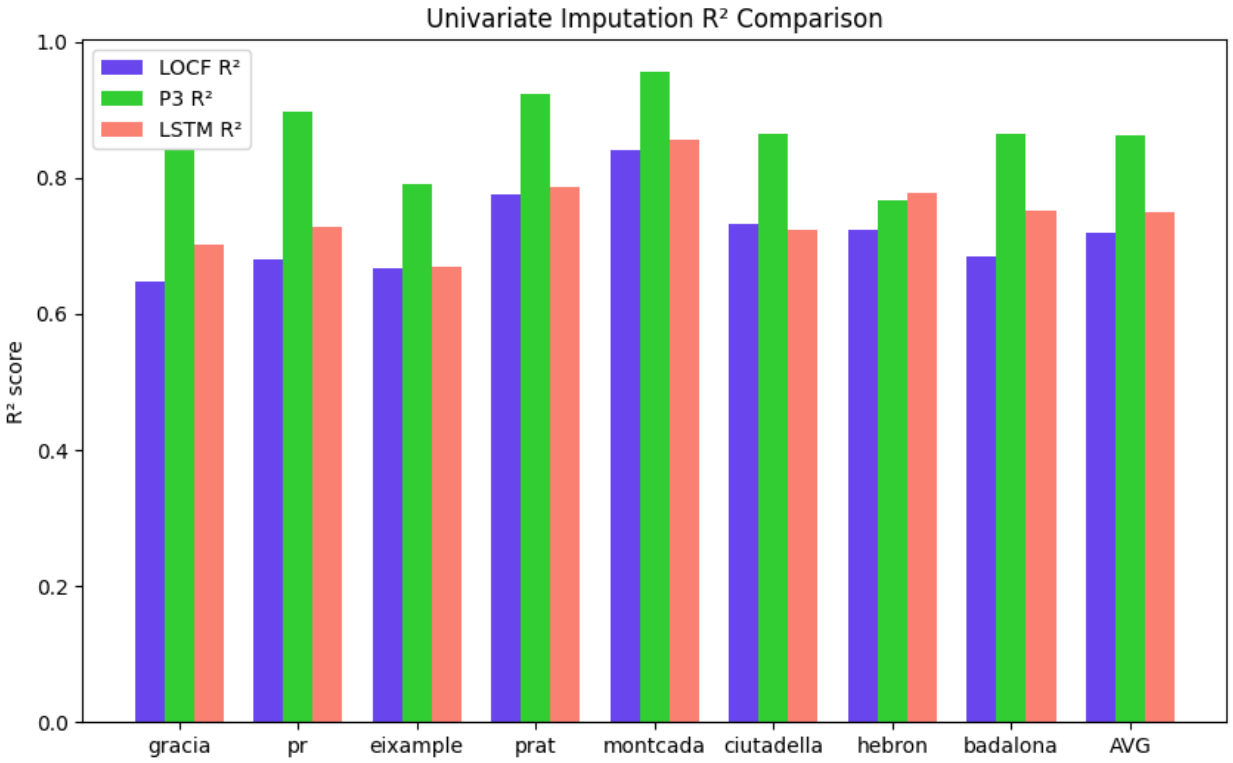

Figure 2: Coefficient of determination of the reconstructed values using 1 sample of `produce missings`.

The average of the results were $MSE = 91.037$ and $R^2 = 0.836$.

LSTM

Using `pytorch` we defined a RNN to predict new values from a window of T past values. If a prediction is attempted for a position $i < T$ in the dataset, we set artificially set the first $T - i$ values of the windows as 0s. The idea was to build a single model that would be train on the data of all districts. We defined this custom class for the model:

```
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim):
        super(LSTMModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])
        return out
```

We did some data scaling to the range $[0, 1]$ with `sklearn.preprocessig.MinMaxScaler` just for this model, and used `CUDA` to accelerate the training on a GPU, but the model was still much slower than the previous ones. We tried several choices for the hyperparameters, and the results always turned out to be worse than the ones with polynomial interpolation.

For these reasons, we considered that it was not worth it to perform any advanced hyperparameter selection techniques, and ended up choosing `hidden_dim=10, num_layers=2` and $T = 10$. The training process was a straightforward optimization using `Adam` with a learning rate of 0.001, for 50 epochs.

This model was by far the most difficult to implement, and the heaviest in terms of memory and time costs. Its performance was just a tiny nudge better than the baseline LOCF, which was very disappointing.

The average of the results were $MSE = 167.250$ and $R^2 = 0.753$.

Multivariate Models

Now we consider models that use the signals from all 8 sensors to fill the missing values. As in the previous section, we will only discuss the results for a single instance of the `produce missings` function. Figures

MICE using Multiple Linear Regression

We used the `IterativeImputer` method from `sklearn.impute`, setting `estimator=LinearRegression(), max_iter = 200, to`
As with all the methods from `sklearn`, it was very easy to implement, and the performance was quite good.

The average of the results were $MSE = 136.246$ and $R^2 = 0.816$.

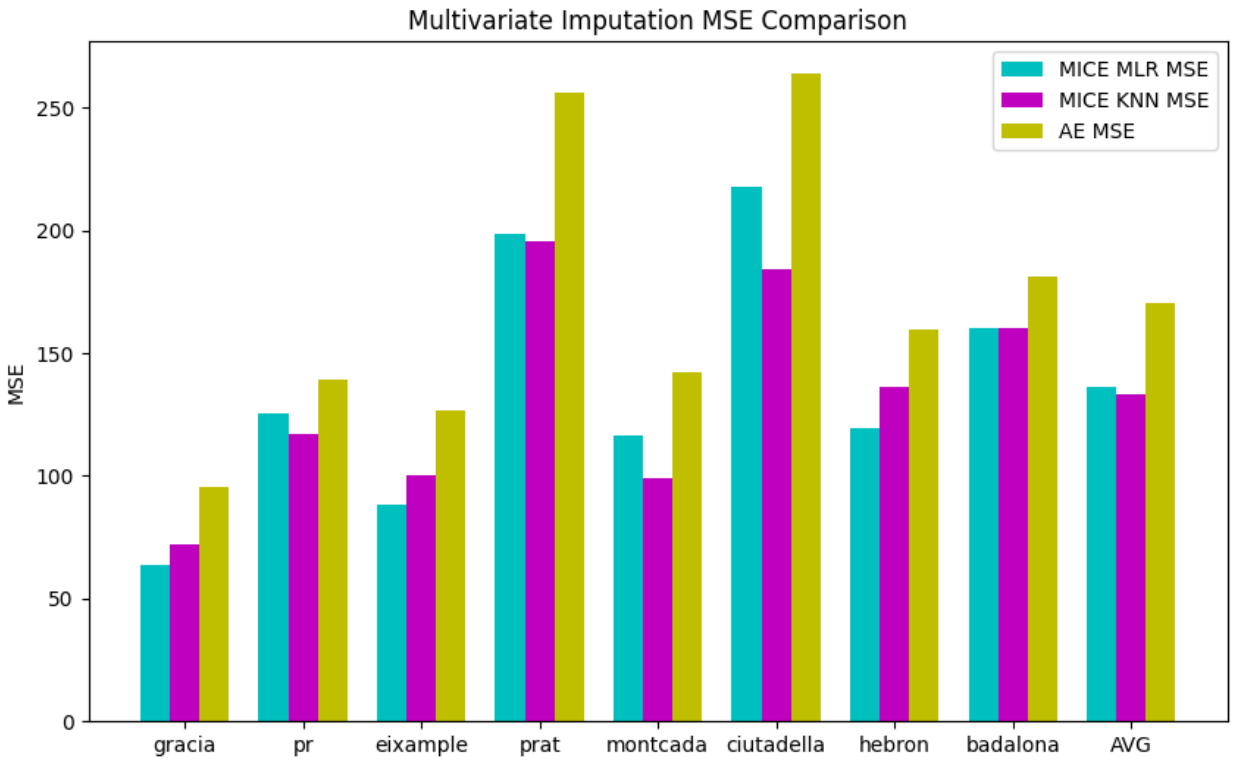

Figure 3: Multivariate Models MSE on the given instance

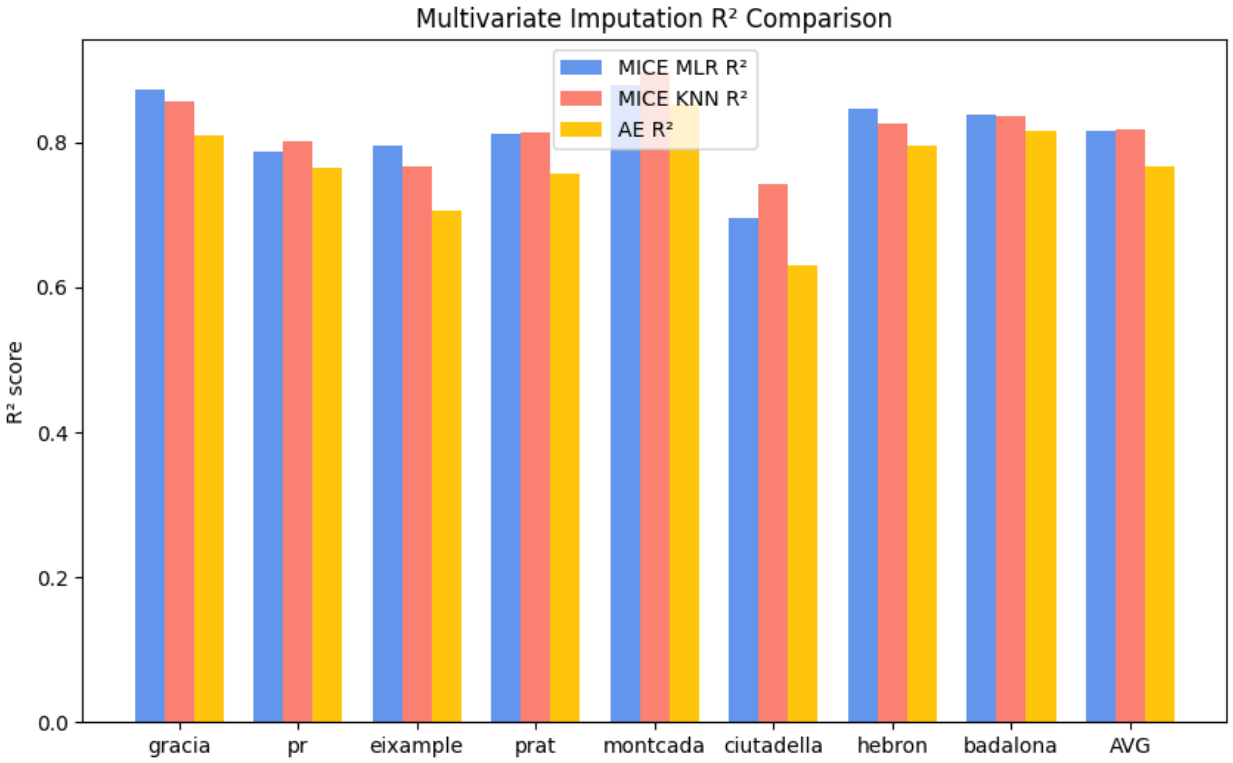

Figure 4: Multivariate Models R squared score on the given instance

MICE using KNN

We also considered the same Multiple Imputation By Chain Equation with `estimator = KNeighborsRegressor(n neighbors=18)`. The value for `n_neighbors` was found using a grid search, that is, by exhaustive trial and error. There was a slight accuracy improvement with respect to the MLR estimator, but the running time was approximately 30 times slower.

The average of the results were $MSE = 118.717$ and $R^2 = 0.832$.

Auto Encoder

Using `pytorch`, we built an undercomplete AE with the following architecture:

```
class Autoencoder(nn.Module):
def __init__(self):
    super(Autoencoder, self).__init__()
    self.encoder = nn.Sequential(
        nn.Linear(8, 6),  # Increased complexity
        nn.ReLU(True),
        nn.Linear(6, 4),  # Increased complexity
        nn.ReLU(True),
        nn.Linear(4, 2)  # Increased latent space dimension
    )
    self.decoder = nn.Sequential(
        nn.Linear(2, 4),
        nn.ReLU(True),
        nn.Linear(4, 6),
        nn.ReLU(True),
        nn.Linear(6, 8),
    )

def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x
```

As expected, the model was prone to underfitting, which is the reason why we chose to increase the depth of each neural network to 2. The model takes a measurement of all 8 sensors, and returns a decoded-encoding of the measurement.

Using the dataset with missing values to train this model was difficult, as an input with just 1 `NaN` value becomes intractable. Because of that, we decided to train the model using the original dataset. Naturally, this should be giving an unfair advantage to the MVI using this model, but in the end our results pointed the contrary.

Using minibatch gradient descent, we were able to train a model achieving a mean-square error loss of 91.995.

Then, with that model we performed the iterative MVI method proposed in the assignment. We took of from a dataset with mean value imputation, and substituted the values of the missing positions by the forward evaluations of the AE. The sum of absolute differences between one iteration and the next became less than $1e-3$ after 132 iterations.

The convergent results were considerably less accurate than the ones obtained with the MICE methods, even though we used the original data to train the AE.

The average of the results were $MSE = 170.565$ and $R^2 = 0.766$.

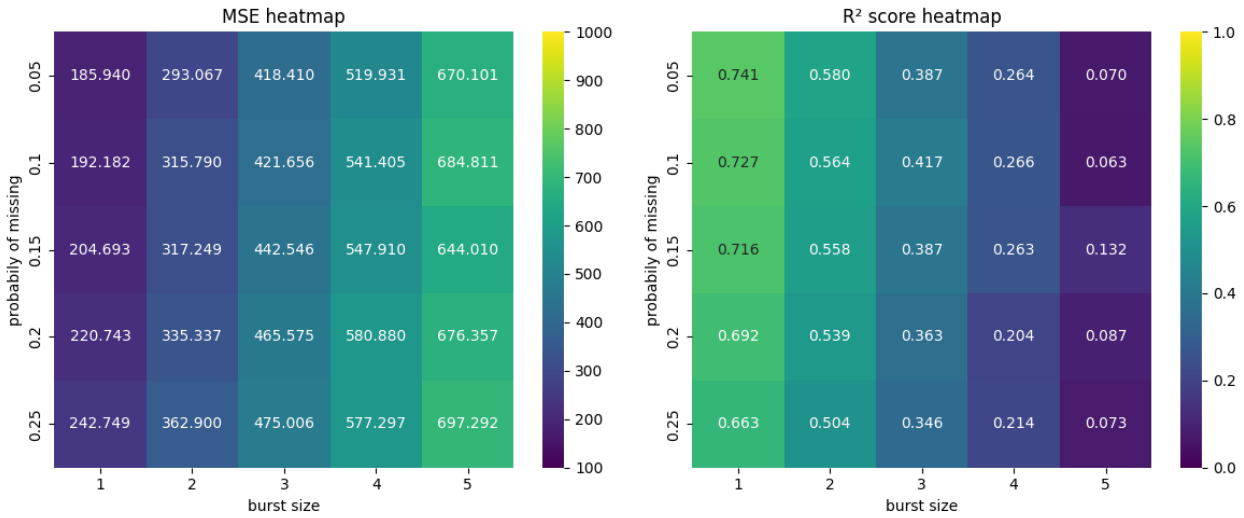

Figure 5: Accuracy Heatmaps for the LOCF method.

Statistical comparison and Conclusions

To ensure that our results have statistical sense, we need to test them on several instances of the `produce_missings` function. We perform a kind of grid-search on that function parameters.

We explored a grid of $p = [0.05, 0.1, 0.15, 0.2, 0.25]$ for the percentage of desired missings, and $b = [1, 2, 3, 4, 5]$ for the bursts of missing values. For each pair of parameters, we performed 10 different instances of `produce_missings` function (as it is non-deterministic), and then took the averages of the MVI of all instances.

It was decided to exclude the NN methods from this analysis, as they were to computationally intensive and their accuracy was clearly inferior.

Therefore, we only considered the following methods: LOCF (figure 5), polynomial interpolation (figure 6), MICE with MLR (figure 7) and MICE with KNN (figure 8).

The main insights from this analysis are:

- The bursts of missing values size has a great impact on the LOCF method, which is quite intuitive.
- The polynomial interpolation method can be the most accurate, but it is also unstable for large burst sizes.
- The multivariate models are robust to burst size increases.
- The two MICE models have roughly the same accuracy, but the MLR estimator is preferred as it is faster.

In conclusion, the victor for burst sizes of 1 is the polynomial interpolation. For larger burst sizes, a Multivariate model is best.

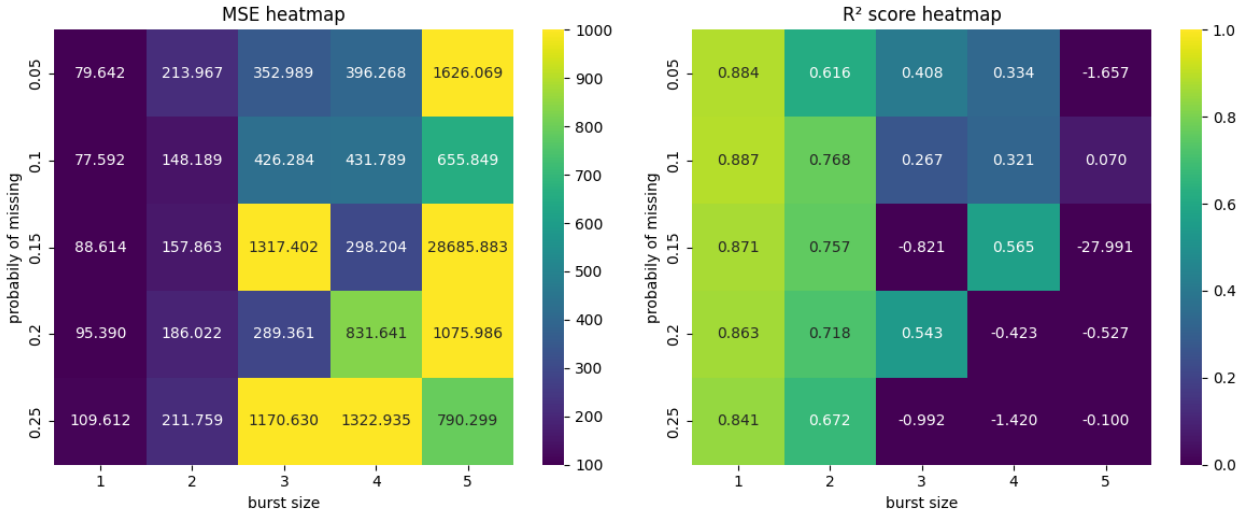

Figure 6: Accuracy Heatmaps for the Cubic Polynomial Interpolation method.

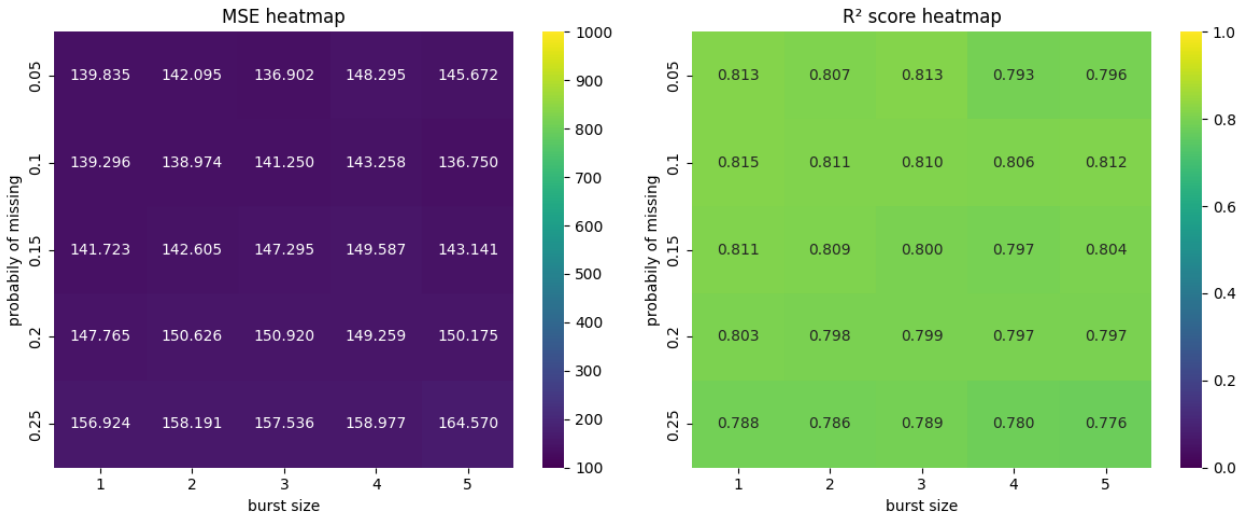

Figure 7: Accuracy Heatmaps for the MICE with MLR method.

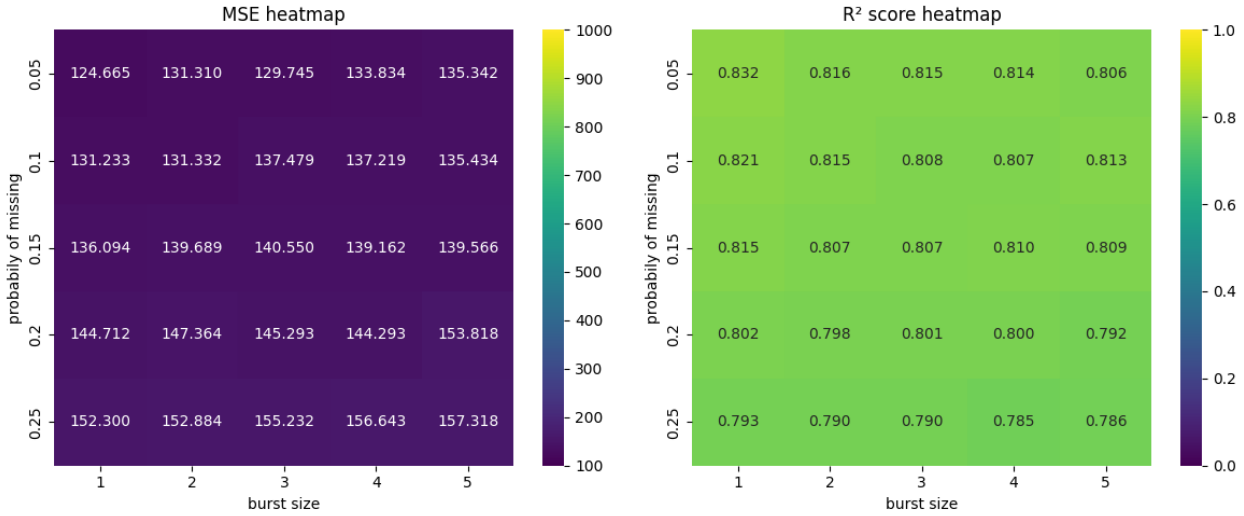

Figure 8: Accuracy Heatmaps for the MICE with KNN method and 18 neighbours.