INTERNATIONAL HELLENIC UNIVERSITY

SCHOOL OF ENGINEERING
DEPARTMENT OF INFORMATION AND ELECTRONIC ENGINEERING

MSc THESIS

«WEBIMPUTER: A Web application for missing value imputation in datasets»

Student
Dimitrios Antoniadis

Supervisor
Stefanos Ougiaroglou
Assistant Professor

September 20, 2023

Τίτλος Δ.Ε. WEBIMPUTER: Web εφαρμογή για καταλογισμό απουσών τιμών σε σύνολα δεδομένων
Κωδικός Δ.Ε. 21190
Ονοματεπώνυμο φοιτητή : Δημήτριος Αντωνιάδης
Ονοματεπώνυμο εισηγητή : Ουγιάρογλου Στέφανος
Ημερομηνία ανάληψης Δ.Ε. : 06-05-2022
Ημερομηνία περάτωσης Δ.Ε.: 20-09-2023

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

«Στους γονείς μου και τον αδερφό μου»

Acknowledgements

At this point, I would like to thank Assistant Professor Stefanos Ougiaroglou for supporting me during the whole period of the masters program and helping me accomplish this thesis. I would also like to thank all the Professors that have been part of the masters program, as they helped me gain a lot of meaningful knowledge in the scientific field of ”web intelligence”. Of course, I couldn’t forget to thank ”my people”, those who have been supporting me all these years, each one in their own way. So, I thank my parents, my brother, my girlfriend, my grandparents and my friends. Thank you from the bottom of my heart!

Abstract

Missing values in datasets is a very important research issue in big data analysis. These datasets are often used for training machine learning models and if a significant percentage of the values is missing, it may result to inaccurate predictions or incorrect model evaluations. To address this issue, several imputation techniques have been proposed as part of the data cleaning process. However, applying these techniques to real-world datasets can be challenging and time-consuming for researchers and data scientists. This thesis presents the development of a web application that utilizes various imputation methods, offering an easy and user-friendly way to handle missing values in datasets. Users can access the website, upload their datasets with missing values in CSV format, choose one of the available imputation methods based on the feature types of the dataset and then download the file containing the imputed values, as soon as the imputation process is complete. The web application, named WEBIMPUTER, offers a variety of imputation solutions for numerical, categorical and mixed feature datasets, providing a wide range of parameter options for the imputation models. Finally, several experiments that have been conducted by applying all the imputation algorithms of the application to various datasets of different file size and measuring the execution time are presented here, to help users gain a better understanding of the computational efficiency of the models.

Table of Contents

Acknowledgements . . . iii
Abstract . . . iv
Table of Contents . . . v
Abbreviations . . . vii

1 Introduction **1**
1.1 Missing Data Imputation . . . 1
1.2 Motivation and Contribution . . . 1
1.3 Thesis Organization . . . 2

2 Imputation Methods **3**
2.1 Mean . . . 3
2.2 Median . . . 3
2.3 Mode . . . 3
2.4 Constant Value . . . 3
2.5 Random Hot-Deck . . . 3
2.6 k-Nearest Neigbors (k-NN) . . . 4
2.7 Mixed k-NN . . . 4
2.8 Naive Bayes . . . 5
2.8.1 Mathematical formulation . . . 5
2.9 Linear Regression . . . 6
2.9.1 Mathematical formulation . . . 6
2.10 Lasso Regression . . . 7
2.10.1 Mathematical formulation . . . 7
2.11 Ridge Regression . . . 8
2.11.1 Mathematical formulation . . . 9
2.12 Elastic-Net Regression . . . 10
2.12.1 Mathematical Formulation . . . 10
2.12.2 Advantages . . . 11
2.12.3 Challenges . . . 12
2.13 Support Vector Regression (SVR) . . . 12
2.13.1 Mathematical Formulation . . . 14

3 WEBIMPUTER implementation **16**
3.1 Technologies Used . . . 16
3.1.1 Front-end . . . 16
3.1.2 Back-end . . . 16
3.1.3 Python libraries . . . 16
3.2 Application Architecture . . . 17
3.3 Imputation Modules . . . 18
3.3.1 simple_imputer.py . . . 18
3.3.2 regression_imputer.py . . . 21
3.3.3 svr_imputer.py . . . 24
3.3.4 naive_bayes_imputer.py . . . 27
3.3.5 random_hot_deck_imputer.py . . . 29
3.3.6 numerical_knn_imputer.py . . . 30
3.3.7 categorical_knn_imputer.py . . . 32
3.3.8 mixed_knn_imputer.py . . . 35
3.3.9 dataset_validator.py . . . 38
3.4 JavaScript Modules . . . 40
3.4.1 prevent_upload.js . . . 40
3.4.2 show_available_methods_and_parameters.js . . . 44
3.4.3 show_download_link.js . . . 51

4 WEBIMPUTER Presentation **53**
4.1 Home Page . . . 53
4.2 Imputer Page . . . 53
4.3 Imputation Methods Page . . . 54
4.4 About Page . . . 55
4.5 Success Response . . . 55
4.6 Validation and Errors . . . 55
4.6.1 Invalid File Format . . . 56
4.6.2 Invalid Model Parameters . . . 56
4.6.3 Dataset Exceeds Maximum Size . . . 57
4.6.4 Imputation Still in Progress . . . 58
4.6.5 Imputation Requires More Time . . . 59

5 WEBIMPUTER Experiments and Metrics **60**
5.1 Numerical Datasets . . . 60
5.1.1 Mean . . . 60
5.1.2 Median . . . 61
5.1.3 Random Hot-Deck . . . 61
5.1.4 k-NN (Mean) . . . 61
5.1.5 Linear Regression . . . 62
5.1.6 Lasso Regression . . . 62
5.1.7 Ridge Regression . . . 62
5.1.8 Elastic-Net Regression . . . 63
5.1.9 Support Vector Regression (SVR) . . . 63
5.1.10 Methods Comparison . . . 68
5.2 Categorical Datasets . . . 68
5.2.1 Mode . . . 68
5.2.2 Random Hot-Deck . . . 69
5.2.3 k-NN (Mode) . . . 69
5.2.4 Naive Bayes . . . 69
5.2.5 Methods Comparison . . . 69
5.3 Mixed Datasets . . . 70
5.3.1 Mean for Numerical - Mode for Categorical . . . 70
5.3.2 Median for Numerical - Mode for Categorical . . . 70
5.3.3 Random Hot-Deck . . . 71
5.3.4 Mixed k-NN (Mean for Numerical - Mode for Categorical) . . . 71
5.3.5 Mixed k-NN (Median for Numerical - Mode for Categorical) . . . 72
5.3.6 Methods Comparison . . . 72

6 Conclusion and Future Work **73**
6.1 Conclusion . . . 73
6.2 Future Work . . . 73

References **74**

Abbreviations

Δ.Ε. Διπλωματική Εργασία
ΔΙΠΑΕ Διεθνές Πανεπιστήμιο Ελλάδος

Chapter 1: Introduction

1.1 Missing Data Imputation

In real-world data, missing values are frequently present. Missing values in datasets usually result from human error during data collection or from damaged data, and if those missing observations are removed, the data will become biased. Missing data can also have an impact on training machine learning models, resulting in less accurate predictions or classifications. To avoid that, missing values can be imputed using statistical and machine learning techniques so that the maximum amount of information is recovered while maintaining the objectivity of the data.

Anytime data scientists or researchers run into missing value issues in their work, they should first look at the missing values' patterns. Little and Rubin in their book "Statistical Analysis with Missing Data" (1987) identified three categories for missing patterns: "Missing Completely at Random", "Missing at Random" and "Missing Not at Random".

"Missing completely at random" (MCAR), which occurs when missing values have no dependency on any other variable.

"Missing at random" (MAR), which occurs when missing values depend on other variables. In this case, the missing values can be estimated using other variables.

"Missing not at random" (MNAT), which occurs when missing values depend on other missing values, and thus missing data cannot be estimated from existing variables. [1]

1.2 Motivation and Contribution

In many data analysis domains, missing data is a typical issue. Incomplete datasets can be caused by a variety of factors, such as measurement error or data input issues, and can significantly affect the validity and dependability of machine learning models trained using this data. There are a lot of imputation techniques that can help solve this issue, both statistical such as mean and mode imputation, and machine learning such as k-nn, naive bayes and neural network imputation.

Despite the existence of these methods, imputation of missing data is still a difficult and time-consuming operation, especially for users who are unfamiliar with statistical and machine learning methodologies and huge datasets. Therefore, a web application that implements missing value imputation could be very useful and time-saving.

This thesis aims to build a web application for missing value imputation that can be used by both novice and expert users (e.g. data scientists). Users will be able to access the website, upload their missing value dataset in csv format, choose one of the available imputation methods, and then download the imputed dataset once the imputation process is complete.

1.3 Thesis Organization

This thesis consists of six chapters. In Chapter 1 an introduction to missing data is made and its importance of handling and imputing these data is highlighted. After that, the motivation and contribution of the thesis is described, followed by the its overview and structure.

In Chapter 2 the imputation methods that have been used in the web application are presented and their mathematical formulations are analyzed.

In Chapter 3 the WEBIMPUTER implementation is described. That includes the technologies that have been used (programming languages, frameworks and libraries), the software architecture and the analysis of the modules that have been created and implement the various imputation algorithms.

In Chapter 4 the complete WEBIMPUTER application is presented. In this chapter the user can learn how to use the website step by step.

In Chapter 5 some experiments that have been made and metrics that have been monitored are presented.

In Chapter 6 the conclusions of the thesis are stated. Finally some of the most important future work is suggested here.

Chapter 2: Imputation Methods

2.1 Mean

This is a simple statistical method that can be used in numerical data. It calculates the mean of all values in the feature column that are not null and then replaces all the missing values in that column by that value [2]. The formula for the mean imputation is the following:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

,where n is the total number of non-null values in the feature column and x_i is the i-th non-null value.

2.2 Median

This statistical method is also used in numerical datasets. It calculates the median of all values in the feature column that are not null and then replaces all the missing values in that column by that value [2]. First a sorting of the non-null values is made and then the following formula for the median imputation is applied:

$$\text{median} = \begin{cases} x_{\frac{n+1}{2}} & \text{if } n \text{ is odd} \\ \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}) & \text{if } n \text{ is even} \end{cases} \tag{2}$$

,where n is the total number of non-null values in the feature column $x_{(n+1)/2}$ is the middle non-null value and $x_{n/2} + x_{n/2+1}$ is the sum of the two middle non-null values.

2.3 Mode

This statistical method can be applied both to numerical and categorical datasets. It calculates the most frequent element of the non-null values column and and then replaces all the missing values in that column by that value.

2.4 Constant Value

This method simply replaces all the null values in the feture column by a constant value given by the user.

2.5 Random Hot-Deck

In this statistical method the missing values are replaced with values that are drawn randomly from a set of non-null values. This set is called donor set and as in all hot-deck methods it is part of the same dataset. The set that contains the missing values and receives values from the donor set is called recipient set [3].

2.6 k-Nearest Neigbors (k-NN)

This machine learning method that is used in numerical datasets imputes the missing values by finding the k nearest neighbors that don't have null values for every record that contains missing data [4]. To achieve this, first all the distances between a record with missing data and the records with non-missing data are calculated. Then the average of the k records with the smallest distance is calculated and this value is used to impute the missing data record. Two typical distance types are the Euclidean and the Manhattan distance. The formula for the Euclidean distance is the following:

$$d_{euclidean} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 + \cdots + (n_i - n_j)^2} \tag{3}$$

The formula for the Manhattan distance is the following:

$$d_{manhattan} = |x_i - x_j| + |y_i - y_j| + |z_i - z_j| + \cdots + |n_i - n_j| \tag{4}$$

In both formulas $x_i, y_i, z_i ... n_i$ are the non-null features of a missing data record and $x_j, y_j, z_j ... n_j$ are the corresponding features of a non-missing data record. It should be noted here that all the numerical values that are used in the calculation of the Euclidean or the Manhattan distance are normalized first, in order to be between 0 and 1. This way, the effect of the units of measurement and the order of magnitude of the features is diminished. In this thesis, the vector normalization for each feature was used, given by the following formula:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|} \tag{5}$$

,where **v**=[v_1 v_2 v_3 $\ldots v_n$] is the feature and its (non-null) values and $\|\mathbf{v}\|$ is its norm:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + v_3^2 + \cdots + v_n^2} \tag{6}$$

The k-Nearest Neigbors method is one of the most popular and effective imputation methods. However it should be noted that for very large dataset it might be resource and time consuming, as it requires the distance calculation of all missing data records with all the non-missing data records. So depending on the size of the dataset and the demands for the imputation, it should be whether this method should be applied or not.

2.7 Mixed k-NN

This is a variation of the k-NN method, in order to be applied to datasets that have also categorical data. This method also uses the Euclidean or the Manhattan distance formula for the numerical data. However, for the categorical data, the Hamming distance is used:

$$d_{\text{Hamming}} = \begin{cases} 0 & \text{if } x_i = x_j \\ 1 & \text{if } x_i \neq x_j \end{cases} \tag{7}$$

,where x_i is a categorical feature value from a missing data record and x_j is the corresponding categorical

value of a non-missing data record. It should be noted again that all the numerical values that are used in the calculation of the Euclidean or the Manhattan distance are normalized first, in order to be between 0 and 1. This way, not only the effect of the units of measurement and the order of magnitude of the features is diminished but also the Euclidean or the Manhattan distance becomes comparable with the Hamming distance.

2.8 Naive Bayes

The Naive Bayes imputation algorithm can be used in categorical datasets. It uses the Naive Bayes classifier as a tool for predicting the missing values of a feature, making the assumption that each feature contributes equally and independently to the outcome [5].

The Naive Bayes method in categorical datasets is described as follows:

First, the prior probabilities of each possible value for the missing features are calculated. The prior probability is the probability of a particular value occurring in the dataset, indipendently of the other features.

After that, the likelihood is calculated, which is the probability of observing the other features in the dataset, given a particular value for the missing feature.

Finally, using Bayes theorem the algorithm combines the prior probability and the likelihood to calculate the posterior probability of each possible value for the missing feature. The posterior probability is the probability of each possible value for the missing feature, given the values of the other features in the dataset.

The value with the highest posterior probability is selected as the imputed value for the missing feature [6].

2.8.1 Mathematical formulation

Suppose we have a dataset with n features $X_1, X_2, ..., X_{j-1}, X_{j+1}, ..., X_n$, and we want to impute the missing value for feature X_j. We denote the value of X_j as x_j, and the values of the other features as $x_1, x_2, ..., x_{j-1}, x_{j+1}, ..., x_n$. We also assume that the dataset has m data points.

The formula for Naive Bayes imputation is:

$$\begin{aligned} &P(X_j = x_j | X_1 = x_1, X_2 = x_2, \ldots, X_{j-1} = x_{j-1}, X_{j+1} = x_{j+1}, \ldots, X_n = x_n) \\ &= \frac{1}{Z} P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n | X_j = x_j) P(X_j = x_j) \end{aligned} \tag{8}$$

Where:

$P(X_j = x_j | X_1 = x_1, X_2 = x_2, ..., X_{j-1} = x_{j-1}, X_{j+1} = x_{j+1}, ..., X_n = x_n)$ is the conditional probability of the missing value of X_j given the values of the other features. $P(X_1 = x_1, X_2 = x_2, ..., X_n = x_n | X_j = x_j)$ is the joint probability of the values of all the features, given the value of X_j. $P(X_j = x_j)$ is

the prior probability of X_j and Z is a normalization constant that ensures that the sum of all probabilities is 1.

To impute the missing value for X_j, the conditional probabilities for all possible values of x_j are calculated and the value with the highest probability is chosen.

2.9 Linear Regression

Linear regression is a statistical modeling technique that is used to estimate the relationship between a dependent variable and one or more independent variables. In simple linear regression, there is only one independent variable, whereas in multiple linear regression, there are two or more independent variables.

The goal of linear regression is to find the best-fit line that can predict the value of the dependent variable based on the values of the independent variables. The best-fit line is determined by minimizing the sum of the squared differences between the predicted and actual values [7].

Linear regression can be used for imputing missing values because it can estimate the relationship between the target variable and the other variables in the dataset. If the target variable has missing values, we can use linear regression to predict the missing values based on the other variables in the dataset.

2.9.1 Mathematical formulation

The linear regression model can be expressed as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon \tag{9}$$

Where:

$$\begin{gathered} y \text{ is the dependent variable (target),} \\ \beta_0, \beta_1, \beta_2, \ldots, \beta_p \text{ are the coefficients (parameters) to be estimated,} \\ x_1, x_2, \ldots, x_p \text{ are the independent variables (features),} \\ \varepsilon \text{ represents the error term.} \end{gathered}$$

The goal of linear regression is to estimate the coefficients $\beta_0, \beta_1, \ldots, \beta_p$ that minimize the residual sum of squares (RSS):

$$\text{minimize: RSS} = \sum_{i=1}^{n} \epsilon_i^2 = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}))^2 \tag{10}$$

Once these coefficients are determined, the model can be used for predicting the missing value of a feature given the values of the other non-missing features as input.

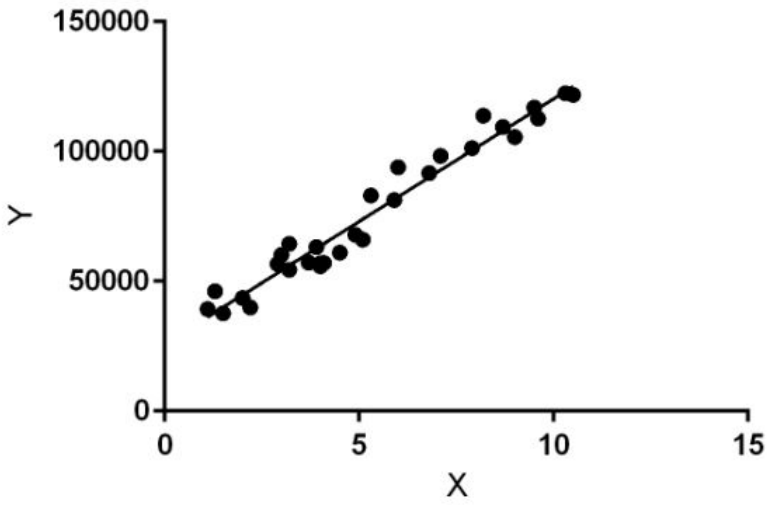

Figure 2.1: Linear regression

2.10 Lasso Regression

Lasso regression is a regularization technique, an extension of ordinary least squares (OLS) regression. It is used over regression methods for a more accurate prediction. This model uses shrinkage, meaning data values are shrunk towards a central point as the mean, which leads to simple, sparse models (models with fewer parameters). This particular type of regression is well-suited for models showing high levels of multicollinearity or certain parts of model selection needs to be automated, like variable selection/parameter elimination [8].

Lasso Regression uses L1 regularization technique and it is suitable for datasets with a lot of features, because it automatically performs feature selection.

2.10.1 Mathematical formulation

Lasso regression starts with a standard linear regression model as described in the previous section, but it introduces an additional penalty term (L1 regularization) based on the absolute values of the coefficients. The L1 regularization term is the sum of the absolute values of the coefficients multiplied by a tuning parameter α:

$$L_1 = \alpha \left(|\beta_1| + |\beta_2| + \ldots + |\beta_p| \right) \tag{11}$$

Where:

α is the regularization parameter that controls the amount of regularization applied,

$\beta_1, \beta_2, \ldots, \beta_p$ are the coefficients.

The objective of Lasso regression is to find the values of the coefficients that minimize the sum of squared differences between the predicted values and the actual values while also minimizing the L1 regularization

term:

$$\text{minimize: RSS} + L_1 \tag{12}$$

Where:

RSS is the residual sum of squares (the error between the predicted values and the actual values).

By adding the L1 regularization term, Lasso regression can shrink the coefficients towards zero. When α is sufficiently large, some coefficients are driven to exactly zero. This property of Lasso makes it useful for feature selection, as variables with zero coefficients are effectively removed from the model.

The choice of the regularization parameter α is crucial in Lasso regression. A larger α value increases the amount of regularization, leading to more coefficients being pushed towards zero. Conversely, a smaller α value reduces the regularization effect, allowing more variables to have non-zero coefficients.

To estimate the coefficients in Lasso regression, an optimization algorithm is used to minimize the objective function. Coordinate descent is commonly used here, which iteratively updates each coefficient while holding the others fixed.

Lasso regression offers a powerful framework for both prediction and feature selection, especially when dealing with high-dimensional datasets where the number of features is large. By finding a balance between simplicity and accuracy, Lasso can provide interpretable models while effectively managing the risk of overfitting.

2.11 Ridge Regression

Ridge regression is an extension of ordinary least squares (OLS) regression, commonly used to fit a linear relationship between independent and dependent variables, which is designed to handle scenarios where predictor variables exhibit high collinearity or strong correlation [8].

When multicollinearity exists, traditional regression models may lead to inconsistent or unreliable results. Ridge regression addresses this issue by adding a regularization term to the objective function, which penalizes large coefficient values. This penalty encourages the model to distribute the impact of correlated variables more evenly, reducing their dominance.

By finding a balance between model complexity and data fitting, Ridge regression produces more stable and accurate predictions, effectively mitigating the problems associated with multicollinearity.

2.11.1 Mathematical formulation

Ridge regression starts with a standard linear regression model as described in the previous section, but it introduces an additional penalty term (L2 regularization):

$$\text{L2} = \alpha \sum_{j=1}^{p} \beta_j^2 \tag{13}$$

Where:

α is the regularization parameter and
β_j are the coefficients to be estimated.

This regularization term encourages small values of the coefficients, helping to prevent overfitting and improve model stability by penalizing large coefficient values. The choice of α determines the strength of regularization iand it is often selected using techniques like cross-validation to find the value that minimizes the prediction error in a validation dataset.

By adding this regularization term, Ridge's objective function that should be minimized takes the form:

$$\text{Minimize: } J(\beta) = \sum_{i=1}^{N} (y_i - y_i')^2 + \alpha \sum_{j=1}^{p} \beta_j^2 \tag{14}$$

Minimizing this objective function is typically done by taking the derivative of $J(\beta)$ with respect to β and setting it to zero:

$$\frac{\partial J(\beta)}{\partial \beta} = 0 \tag{15}$$

The closed-form solution for the above equation is given by:

$$\hat{\beta}^{ridge} = (X^T X + \lambda I)^{-1} X^T y \tag{16}$$

Where:

$\hat{\beta}^{ridge}$ represents the estimated ridge regression coefficients.
X is the matrix of predictor variables.
y is the vector of target values.
I is the identity matrix.

To make predictions using the Ridge regression model, the estimated coefficients ($\hat{\beta}^{ridge}$) can be used as follows:

$$y' = X\hat{\beta}^{ridge} \tag{17}$$

2.12 Elastic-Net Regression

Elastic-Net regression is a regularization technique, an extension of ordinary least squares (OLS) regression, which combines the advantages of Lasso and Ridge regression. It can handle multicollinearity, reduce overfitting, and select relevant features. But it also has some drawbacks and challenges that a researcher or a data scientist should be aware of.

Elastic-Net regression, like Lasso and Ridge uses a penalty term to shrink the coefficients of the predictors. The penalty term is a combination of the L1 norm and the L2 norm of the coefficients described in previous sections, and includes a regularization hyperparameter, called alpha (α) weighted by a parameter called 'L1_ratio'. The L1 norm penalty is similar to Lasso, which tends to produce sparse linear models by setting some coefficients to zero. The L2 norm penalty is similar to Ridge, which tends to reduce the variance of the coefficients by shrinking them towards zero.

The hyperparameter alpha (α) controls the overall strength of regularization. A large value of α means strong regularization, which can lead to less features being selected and thus simpler models. A small value of α reduces the strength of regularization, allowing the model to fit the data more closely, which can lead to more features being selected and thus more complex models with larger coefficients. When α is zero the regularization term is removed and the model becomes equivalent to Ordinary Least Squares (OLS) regression [9].

The hyperparameter L1_ratio takes values in range:

$$0 \leq \text{L1\_ratio} \leq 1 \tag{18}$$

When L1_ration is zero, Elastic-Net is equivalent to Ridge (L2 norm). When L1_ration is one, elastic net is equivalent to Lasso (L1 norm). When L1_ration is between zero and one, Elastic-Net is a compromise between Lasso and Ridge regression. This makes Elastic-Net a flexible method that can adapt to different situations and datasets.

2.12.1 Mathematical Formulation

By adding terms alpha (α) and L1_ratio, the objective function that should be minimized takes the form:

$$\text{minimize: } J(w) = \frac{1}{2n_{\text{samples}}} \|y - Xw\|_2^2 + \alpha \,\text{L1\_ratio}\, \|w\|_1 + \frac{1}{2}\alpha\,(1 - \text{ L1\_ratio})\, \|w\|_2^2 \tag{19}$$

Where:

n_{samples} is the number of training samples.
X is the matrix of input features.
y is the vector of target values.
w is the vector of coefficients to be calculated.
α is the regularization strength parameter.
L1_ratio is the mixing parameter between L1 and L2 regularization.

The closed-form solution for the vector of coefficients w, which obtained by setting the derivative of the loss function $J(w)$ to zero is the following:

$$w = \left(X^T X + \alpha(1 - \text{L1\_ratio})I\right)^{-1} \left(X^T y - \alpha \text{L1\_ratio sign}(w)\right) \tag{20}$$

Where:

w is the vector of coefficients to be learned.
X is the matrix of input features.
y is the vector of target values.
α is the regularization strength parameter.
L1_ratio is the mixing parameter between L1 and L2 regularization.
I is the identity matrix.
sign(w) represents the sign of the coefficients.

Finally, to make predictions using the Elastic-Net regression model, the estimated coefficients (w) can be used as follows:

$$\hat{y} = Xw \tag{21}$$

2.12.2 Advantages

Elastic-Net regression has the advantage of being able to manage multicollinearity, which happens when several predictors have a high degree of correlation with one another. When there is multicollinearity in the dataset, lasso regression can be unstable and inconsistent because it may arbitrarily choose one predictor over another. Ridge regression can handle multicollinearity better, but it may keep too many predictors that are not relevant. Elastic-Net can tackle these problems by selecting a subset of predictors that are correlated, but not redundant.

Another advantage of Elastic-Net regression is that it can reduce overfitting, which occurs when the model

fits the training data too well, but does not perform adequately on a new dataset. Lasso and Ridge can as well reduce overfitting by adding regularization terms, but Elastic-Net is able do it more effectively by combining the benefits of both methods. Elastic-Net regression can balance the bias-variance trade-off by finding the ”golden ratio” between underfitting and overfitting.

A third advantage of Elastic-Net regression is that it is able perform feature selection, which means that the model identifies the most important predictors for the outcome. Lasso regression can also perform feature selection by setting some coefficients to zero, but it may leave out some relevant predictors if there are too many of them. Ridge regression cannot perform feature selection, as it keeps all the predictors, although it shrinks them.

2.12.3 Challenges

One of the challenges of Elastic-Net regression is that it requires tuning two hyperparameters: alpha and L1_ratio. Hyperparameters are parameters that are not learned by the model, but need to be specified by the user. Finding the best values for hyperparameters involves minimizing error or maximizing model performance. Due to the need to test various value combinations and assess their effects, tuning hyperparameters can be time-consuming and computationally expensive.

Elastic-Net regression also has the drawback of occasionally not working well with certain dataset types. When there are far more predictors than observations in high-dimensional data, for instance, Elastic-Net regression may not be appropriate. In this situation, it’s possible that the approach won’t be able to choose the pertinent features or effectively minimize the dimensionality. When there is a non-linear relationship between the variables and the outcome, elastic-net regression may not be acceptable. In this situation, it’s possible that the approach can’t adequately represent the complexity of the data.

A third challenge of Elastic-Net regression is that it may not be easily interpretable or explainable. Interpretability and explainability refer to the ability to understand how the model works and why it makes certain predictions. Lasso and Ridge methods are relatively simple and more intuitive, as they have a clear relationship between the coefficients and the predictors. Elastic-Net, on the other hand, is more complex and ambiguous, as it involves a combination of two penalties and two hyperparameters, which makes it less intuitive. This means that Elastic-Net regression method may not provide a clear or meaningful explanation of the model and its predictions.

2.13 Support Vector Regression (SVR)

Support Vector Machines (SVMs) are a statistical learning theory that was created by Vapnik and Lerner in 1963. In 1996, the theory was further developed by Scholkopf, Burges and Vapnik, who constructed a hyperplane or a set of hyperplanes in a high- or infinite-dimensional space on known data. They successfully used SVM for classification of unseen data. A different version of SVM for regression, which is called support vector regression (SVR), was proposed by Vapnik, Steven Golowich and Alex Smola in 1997. The model produced by support vector classification depends on only a subset of the training data because the cost function for building the model does not consider training points that lie beyond the margin. Analogously, the model produced by SVR depends on only a subset of the training data because

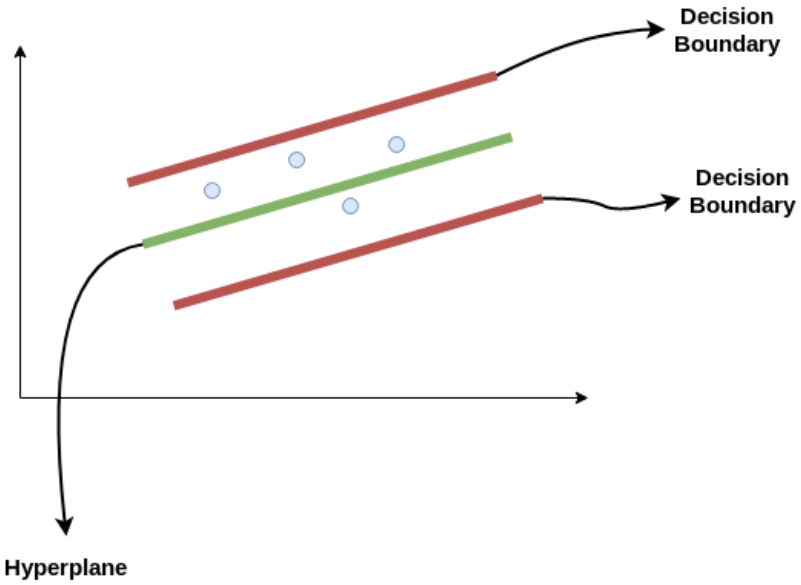

Figure 2.2: Support Vector Regression (SVR)

the cost function for building the model ignores any training data that are close (within a threshold e) to the model prediction. Support vector regression (SVR) is the most commonly used form of support vector machines [10].

The goal of SVR is to find a function that approximates the relationship between the input variables and a continuous target variable, while minimizing the prediction error.

Unlike Support Vector Machines (SVMs) used for classification tasks, SVR seeks to find a hyperplane that best fits the data points in a continuous space. This is achieved by mapping the input variables to a high-dimensional feature space and finding the hyperplane that maximizes the margin (distance) between the hyperplane and the closest data points, while also minimizing the prediction error.

SVR can handle non-linear relationships between the input variables and the target variable by using a kernel function to map the data to a higher-dimensional space. This makes it a powerful tool for regression tasks where there may be complex relationships between the input variables and the target variable.

Support Vector Regression (SVR) uses the same principle as SVM, but for regression problems. The problem of regression is to find a function that approximates mapping from an input domain to real numbers on the basis of a training sample.

Consider these two red lines shown in figure 2.2 as the decision boundary and the green line as the hyperplane. The objective, when moving on with SVR, is to basically consider the points that are within the decision boundary line. The best fit line is the hyperplane that has a maximum number of points.

Consider the decision boundary lines being at any distance, say ’ε’, from the hyperplane. So, these are the lines that we draw at distance ’+ε’ and ’-ε’ from the hyperplane. This ’ε’ is referred as epsilon.

Assuming that the equation of the hyperplane is as follows:

$$Y = wx + b \tag{22}$$

Then the equations of decision boundary become:

$$wx + b = +\epsilon \tag{23}$$

$$wx + b = -\epsilon \tag{24}$$

Thus, any hyperplane that satisfies our SVR should satisfy:

$$-\epsilon < Y - wx + b < +\epsilon \tag{25}$$

The main aim is to decide a decision boundary at 'ε' distance from the original hyperplane such that data points closest to the hyperplane or the support vectors are within that boundary line.

Hence, only those points that are within the decision boundary and have the least error rate or are within the margin of tolerance are being taken.

SVR gives the flexibility to define how much error is acceptable in the model and will find an appropriate line (or hyperplane in higher dimensions) to fit the data.

2.13.1 Mathematical Formulation

In contrast to ordinary least squares (OLS), the objective function of SVR is to minimize the coefficients — more specifically, the l2-norm of the coefficient vector — not the squared error. The error term is instead handled in the constraints, where the absolute error is set less than or equal to a specified margin, called the maximum error, ε (epsilon). Epsilon can be tuned to gain the desired accuracy of the model. The objective function and constraints are as follows:

$$\text{Minimize: } \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}|\xi_i| \tag{26}$$

$$\text{Constraints: } |y_i - w_i x_i| <= \epsilon + |\xi_i| \tag{27}$$

Where:

ξ_i is the slack variable and
C is a hyperparameter.

The concept of the slack variable is that for any value that falls outside of ε, we can denote its deviation from the margin as ξ.

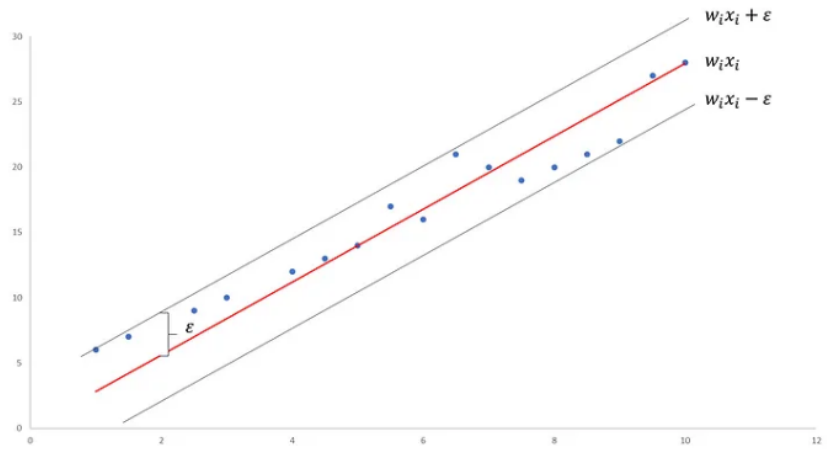

Figure 2.3: Support Vector Regression (SVR)

As the hyperparameter C increases, the tolerance for points outside of ε also increases. As C approaches 0, the tolerance approaches 0 and the equation collapses into the simplified (although sometimes infeasible) one.

Chapter 3: WEBIMPUTER implementation

3.1 Technologies Used

3.1.1 Front-end

The front-end of the application is written in **HTML** with use of **CSS** for styling the pages and **JavaScript** with **jQuery** for adding some functionalities on the client-side. The framework used for the HTML templates and CSS styling is **Bootstrap**.

3.1.2 Back-end

The main functionality of the application is implemented on the server-side. That includes all the modules that implement the imputation algorithms and all the processes that handle the receiving of the datasets with missing values and sending the imputed datasets back to client-side. The datasets that are uploaded by the users should be in CSV format.

The framework used for building the server-side is **Flask** and the programming language is **Python**.

3.1.3 Python libraries

The most important libraries that have been used for the implementation of the imputing algorithms are:

1. **pandas:** It is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools. Pandas has been used in the application for handling the datasets as Data Frames, which is a convenient way to perform complex manipulations and transformations.

2. **NumPy:** It is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays. NumPy has been used in the application for handling multidimensional arrays and perform complex mathematical manipulations and transformations.

3. **sklearn.impute:** It includes modules that perform some popular imputation algorithms such as KNN (for numerical datasets) and some other simple ones such as mean and median imputation.

4. **sklearn.linear_model:** It includes a set of methods intended for regression in which the target value is expected to be a linear combination of the features. In the application it has been used in the implementation of linear, lasso, ridge and elastic-net regression.

5. **sklearn.naive_bayes:** It includes modules for implementing Naive-Bayes classification and it has been used in the application to perform Naive-Bayes imputation in categorical datasets.

6. **sklearn.svm:** It includes modules regarding Support Vector Machines (SVM) and it has been used in the application to perform Support Vector Regression (SVR) imputation.

3.2 Application Architecture

The application architecture and flow is shown in figure 3.1.

1. First the user uploads a dataset in CSV format and defines the following parameters:
   - **Missing Value Representation:** This parameter defines how the missing values are represented in the dataset. Available options are 'NaN', '?' and '-'.
   - **Dataset Features Type:** This parameter defines the type of the dataset according to its features. Available options are 'Numerical' (all features of the dataset are numerical), 'Categorical' (all features of the dataset are numerical) and 'Mixed' (both numerical and categorical data are present in the dataset).
   - **Imputation Method:** This parameter defines the imputation method that will be used. This field is filtered by the 'Dataset Features Type' field (via JavaScript code) so that the user can see only the compatible imputation methods to his/her dataset. The available imputation methods for each dataset type are:
     - **For Numerical Datases:** 'Mean', 'Median', 'Random Hot-Deck', 'k-NN (Mean)', 'k-NN (Median)', 'Linear Regression', 'Lasso Regression' and 'Ridge Regression' and 'Elastic-Net Regression'.
     - **For Categorical Datasets:** 'Mode', 'Random Hot-Deck', 'k-NN (Mode)' and 'Naive Bayes'.
     - **For Mixed Datasets:** 'Mode', 'Mean for Numerical - Mode for Categorical', 'Median for Numerical - Mode for Categorical', 'Random Hot-Deck', 'Mixed k-NN (Mean for Numerical - Mode for Categorical)' and 'Mixed k-NN (Median for Numerical - Mode for Categorical)'.
2. Once the user defines all the above parameters, he/she submits the form and the CSV is uploaded and stored in the server.
3. The appropriate imputation python module is triggered according to the selected method and starts the imputation process asynchronously. That means that the user can continue browsing the application or he/she can upload another dataset in the meantime.
4. Once the imputation process is completed, the imputed dataset is stored in CSV format in the server and a download link is sent asynchronously to the front-end.

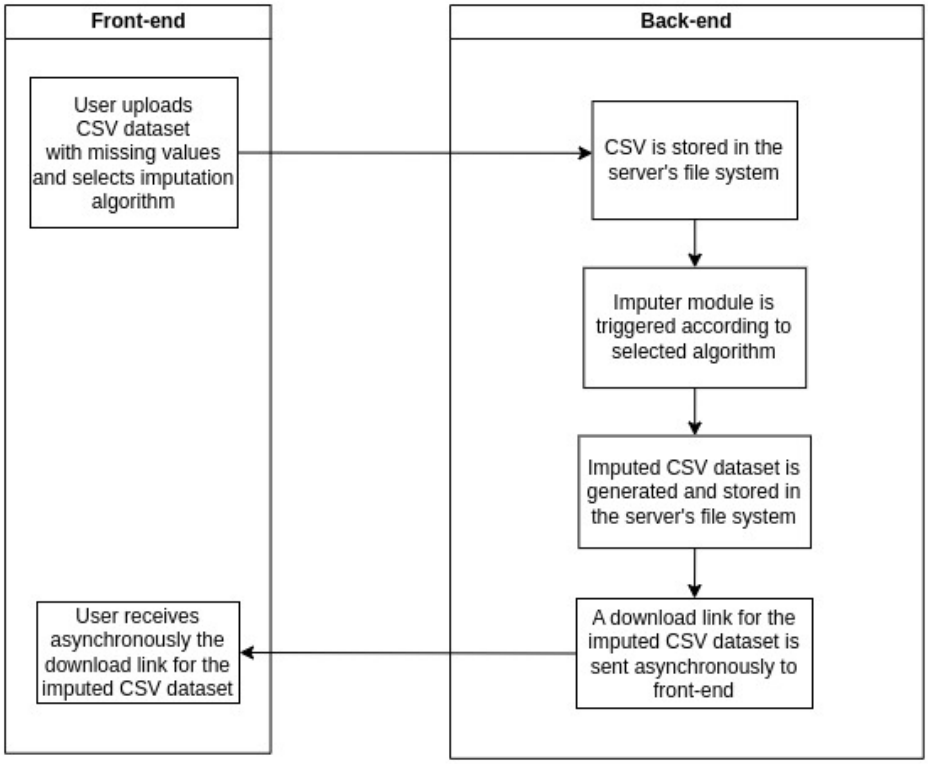

Figure 3.1: WEBIMPUTER architecture

3.3 Imputation Modules

In this section all the python imputation modules are presented and some key points of the algorithms are analyzed.

3.3.1 simple_imputer.py

This module implements the simple imputation methods for numerical, categorical and mixed datasets (datasets that contain both numerical and categorical features). These methods are ’Mean’ and ’Median’ for numerical features and ’Mode’ for categorical. The implementation of these imputing methods is performed using the ’SimpleImputer’ module of ’sklearn.impute’ library.

The imputation algorithm in this module is described as follows:

First the CSV file is read and converted to a pandas Data Frame so that the data handling and the calculations are easier and efficient.

Then a validation is made to ensure that the CSV structure is according to the application’s specifications which are described on the ’About’ page, meaning that numerical features should be declared under their name as ’int’ for integer features, ’float’ for floating point features and ’str’ for categorical/string features.

This check is necessary because the imputation methods should be aware of which types of features are expected to be handled.

After that the given missing value is replaced by 'np.nan' (numpy's 'not-a-number' constant) to prepare the input data in suitable format for the algorithms of 'SimpleImputer'.

Next the Data Frame is devided into separate Data Frames, each one contains the same data type. For instance, a mixed dataset that contains integer, floating point and categorical features will be devided into three separate Data Frames, one that contains the integer features, one that contains the floating point features and one that contains the categorical/string features.

After that for each separate Data Frame a 'SimpleImputer' object is created and is used to impute the missing data, according to the corresponding imputation method ('mean', median', 'mode').

The final step is to merge each and every imputed feature into the original Data Frame and save it in CSV format.

The source code of the module is shown below:

```
1  import logging
2  import shutil
3
4  import numpy as np
5  import pandas as pd
6  from sklearn.impute import SimpleImputer
7
8  from dataset_validator import is_mixed_attribute_types_valid
9
10
11 def simple_imputer(filename, missing_value, imp_method):
12     imp_csv = filename[:-4] + "_imp.csv"
13
14     try:
15         df = pd.read_csv(filename, low_memory=False)
16         attribute_types = df.loc[0, :]
17
18         if not is_mixed_attribute_types_valid(attribute_types, imp_csv):
19             return 1
20
21         df = df.replace(to_replace=missing_value, value=np.nan)
22         int_df = pd.DataFrame()
23         float_df = pd.DataFrame()
24         str_df = pd.DataFrame()
25
26         for i in range(len(attribute_types)):
27             if attribute_types[i] == 'int':  # Alternative is df.iloc[0][i]
28                 int_df[df.columns[i]] = df[df.columns[i]]
29             elif attribute_types[i] == 'float':  # Alternative is df.iloc[0][i]
```

```
                float_df[df.columns[i]] = df[df.columns[i]]
            else:  # 'str'
                str_df[df.columns[i]] = df[df.columns[i]]

        if len(int_df) > 0:
            int_df = int_df.drop([0])
            int_imp = SimpleImputer(missing_values=np.nan, strategy=imp_method)
            int_imp.fit(int_df)
            if imp_method == 'mean':
                int_array_imp = np.rint((int_imp.transform(int_df) + 0.0001))
            else:
                int_array_imp = int_imp.transform(int_df)

            int_array_imp = int_array_imp.astype(int)
            int_df_imp = pd.DataFrame(int_array_imp, columns=int_df.columns)
            int_df_imp = int_df_imp.astype('object')

            for i in range(len(int_df_imp.columns)):
                df[int_df_imp.columns] = int_df_imp[int_df_imp.columns]

        if len(float_df) > 0:
            float_df = float_df.drop([0])
            float_imp = SimpleImputer(missing_values=np.nan, strategy=imp_method)
            float_imp.fit(float_df)
            if imp_method == 'mean':
                float_array_imp = np.round((float_imp.transform(float_df) + 0.0001),
                ↪  decimals=3)
            else:
                float_array_imp = float_imp.transform(float_df)
            float_df_imp = pd.DataFrame(float_array_imp, columns=float_df.columns)
            for i in range(len(float_df_imp.columns)):
                df[float_df_imp.columns] = float_df_imp[float_df_imp.columns]

        if len(str_df) > 0:
            str_df = str_df.drop([0])
            str_imp = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
            str_imp.fit(str_df)
            str_array_imp = str_imp.transform(str_df)
            str_df_imp = pd.DataFrame(str_array_imp, columns=str_df.columns)
            for i in range(len(str_df_imp.columns)):
                df[str_df_imp.columns] = str_df_imp[str_df_imp.columns]

        df.to_csv(imp_csv, index=False)

    except Exception as e:
        logging.error(e)
        error_csv = r'imputation_error.csv'
        shutil.copyfile(error_csv, imp_csv)
        return 1
```

```
78
79      return 0
```

3.3.2 regression_imputer.py

This module implements the linear regression imputation algorithms. The four available algorithms are 'Linear Regression', 'Lasso Regression', 'Ridge Regression' and 'Elastic-Net Regression'. The corresponding models that have been used here are from 'sklearn.linear_model' library and they are 'LinearRegression', 'Lasso', 'Ridge' and 'ElasticNet'.

The imputation algorithm in this module is described as follows:

First the validity of the input "regression_model" parameter is checked. If the input parameter is not one of the available valid options, an Exception is raised. At this point it should be stated that in the context of the application this Exception is expected never to be raised as the regression model input is provided as option in a drop-down box, so that the user cannot provide any invalid input. However it is considered to be a good practice to have this kind of Exceptions, in case of this module will be used in another application in the future.

After that the data from the CSV file are converted to a pandas Data Frame and another validation is made in order to ensure that the dataset has the correct form, meaning that all features should be numerical and declared accordingly ('int' for integer and 'float' for floating point features). Categorical features are not accepted in these regression methods and will lead to error.

Next the Data Frame is split into two Data Frames, one called the 'donor' and the other one called the 'recipient'. The 'donor' Data Frame contains all the records of the dataset that do not have any missing values. The 'recipient' Data Frame, on the other hand, contains all the records of the dataset that have at least one missing value. The 'donor' Data Frame will be used to train and build the regression models which will predict the missing values of the 'recipient' Data Frame.

After this separation the 'recipient' Data Frame is iterated and each of its records gets imputed with the following procedure:

1. For each record the column(s) with the missing feature value(s) is/are stored in the 'missing_attributes' list, which is given as input to the 'make_regression_models' function.

2. This function handles the build of the regression models. First it checks if the model that corresponds to the missing attribute already exists in the 'models_total' list. The 'models_total' list contains all the regression models that have been built so far. This check is done in order to avoid repetitive training and builds of models.

3. If the model exists in the 'models_total' list the function is terminated (returns 0). Otherwise, if the model does not exist, it is built and added in the 'models_total' lis. Depending of the imputation method that has been selected, the corresponding model is built. For example in case of 'elastic_net_regression' the 'ElasticNet' model from 'sklearn.impute' library is selected etc.

4. The model that has been built is used for predicting and imputing the missing values of the record.

Finally the imputed Data Frame is stored in CSV format.

The source code of the module is shown below:

```
import logging
import shutil

import numpy as np
import pandas as pd
from sklearn.linear_model import (
    LinearRegression,
    Lasso,
    Ridge,
    ElasticNet,
)

from dataset_validator import is_numerical_attribute_types_valid

pd.options.mode.chained_assignment = None

VALID_LINEAR_REGRESSION_MODELS = [
    'linear_regression',
    'lasso_regression',
    'ridge_regression',
    'elastic_net_regression',
]

def regression_imputer(filename, missing_value, regression_model='linear_regression',
↪  alpha=1.0, l1_ratio=0.5):
    if regression_model not in VALID_LINEAR_REGRESSION_MODELS:
        raise Exception(f"Invalid linear regression model {regression_model}. It must
        ↪  be one of {', '.join(VALID_LINEAR_REGRESSION_MODELS)}.")

    imp_csv = filename[:-4] + "_imp.csv"

    try:
        df = pd.read_csv(filename, low_memory=False)
        attribute_types = df.loc[0, :]

        if not is_numerical_attribute_types_valid(attribute_types, imp_csv):
            return 1

        cols_num = len(df.columns)
        df = df.drop([0])

        recipient_df = pd.DataFrame()
```

```
        recipient_df = df[(df == missing_value).any(axis=1)]

        donor_df = pd.DataFrame()
        donor_df = df.drop(recipient_df.index)

        missing_attributes_total = []
        models_total = []

        def make_regression_models(miss_attr_list: list):
            if miss_attr_list in missing_attributes_total:
                return 0
            else:
                missing_attributes_total.append(miss_attr_list)
                reg_models_list = []
                df_feat = donor_df.loc[:, ~donor_df.columns.isin(miss_attr_list)]
                arr_feat = df_feat.to_numpy()

                if regression_model == "linear_regression":
                    reg = LinearRegression()
                elif regression_model == "lasso_regression":
                    reg = Lasso(alpha=alpha, max_iter=int(1e6))
                elif regression_model == "ridge_regression":
                    reg = Ridge(alpha=alpha, max_iter=int(1e6))
                else:
                    reg = ElasticNet(alpha=alpha, l1_ratio=l1_ratio,
                    ↪  max_iter=int(1e6))

                for k in range(len(miss_attr_list)):
                    df_target = donor_df.loc[:, donor_df.columns == miss_attr_list[k]]
                    arr_target = df_target.to_numpy()
                    reg.fit(arr_feat, np.ravel(arr_target))
                    reg_models_list.append(reg)
                models_total.append(reg_models_list)
            return 0

        for i in range(len(recipient_df)):
            missing_attributes = []
            for j in range(cols_num):
                if recipient_df.iloc[i, j] == missing_value:
                    missing_attributes.append(recipient_df.columns[j])
            make_regression_models(missing_attributes)
            model_index = missing_attributes_total.index(missing_attributes)
            recip_index = recipient_df.index[i]
            rec_df_target = recipient_df.loc[recip_index,
            ↪  ~recipient_df.columns.isin(missing_attributes)]
            rec_target = [rec_df_target.to_numpy().astype(float)]

            for m in range(len(missing_attributes)):
                prediction = models_total[model_index][m].predict(rec_target)
```

```
88
89                  if attribute_types[missing_attributes[m]] == "float":
90                      recipient_df.loc[recip_index, missing_attributes[m]] =
                        ↪  np.round(prediction[0] + 0.0001, decimals=3)
91                  else:
92                      recipient_df.loc[recip_index, missing_attributes[m]] =
                        ↪  int(np.rint(prediction[0] + 0.0001))
93
94              imp_record = recipient_df.loc[recip_index, :]
95              df.loc[recip_index, :] = imp_record
96
97          df.to_csv(imp_csv, index=False)
98
99      except Exception as e:
100         logging.error(e)
101         error_csv = r'imputation_error.csv'
102         shutil.copyfile(error_csv, imp_csv)
103         return 1
104
105     return 0
```

3.3.3 svr_imputer.py

This module implements the Support Vector Regression (SVR) imputation algorithm. The model that has been used here is 'SVR' from 'sklearn.svm' library.

This algorithm has the same steps and logic as described in section regression imputer (see section 3.3.2). The difference is that the 'make_models' function uses the 'SVR' model to build the models and make the predictions and the imputation of missing values. This model has some parameters that are given by the user. These parameters are:

- **kernel:** Specifies the kernel type to be used in the algorithm. The options are 'RBF', 'Linear', 'Polynomial' and 'Sigmoid'.
- **C:** The regularization parameter. The strength of the regularization is inversely proportional to C. The penalty is a squared L2 norm penalty. Normally it must be strictly positive, however in the application it is restricted in the range [0.001, 1000] due to computational limitations of the server.
- **epsilon:** Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value. Normally it must be non-negative, however in the application it is restricted in the range [0.1, 1000] due to computational limitations of the server.
- **gamma:** Kernel coefficient for kernels RBF, polynomial and sigmoid. If gamma is set to 'auto' (which is the default value) then the function uses $\frac{1}{n_{\text{features}}}$ as value of gamma. If 'scale' is selected the function uses $\frac{1}{n_{\text{features}}\text{Var(X)}}$ as value of gamma. Otherwise, if 'Custom' gamma is selected the user should provide a float value. Normally gamma must be non-negative, however in the application it is restricted in the range [0.001, 1000] due to computational limitations of the server.

- **degree:** The degree of the polynomial kernel function. It is ignored by all other kernels Normally it must be a non-negative integer, however in the application it must be an integer in the range [1, 10] due to computational limitations of the server.
- **coef0:** Independent term in kernel function. It is only significant in polynomial and sigmoid kernels. Normally it can be a float with no other restrictions, however in the application it is restricted in the range [-10, 10] due to computational limitations of the server.

The source code of the module is shown below:

```
import shutil

import numpy as np
import pandas as pd
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR

from dataset_validator import is_numerical_attribute_types_valid

pd.options.mode.chained_assignment = None


def svr_imputer(filename, missing_value, kernel, C=1.0, epsilon=0.1, gamma='auto',
↪  degree=3, coef0=0.0):
    imp_csv = filename[:-4] + "_imp.csv"

    try:
        df = pd.read_csv(filename, low_memory=False)
        attribute_types = df.loc[0, :]

        if not is_numerical_attribute_types_valid(attribute_types, imp_csv):
            return 1

        cols_num = len(df.columns)
        df = df.drop([0])

        recipient_df = pd.DataFrame()
        recipient_df = df[(df == missing_value).any(axis=1)]

        donor_df = pd.DataFrame()
        donor_df = df.drop(recipient_df.index)

        svr_missing_attributes = []
        svr_models = []

        def make_svr_models(miss_attr_list: list):
```

```
            if miss_attr_list in svr_missing_attributes:
                return 0
            else:
                svr_missing_attributes.append(miss_attr_list)
                svr_models_list = []
                df_feat = donor_df.loc[:, ~donor_df.columns.isin(miss_attr_list)]
                arr_feat = df_feat.to_numpy()

                for k in range(len(miss_attr_list)):
                    df_target = donor_df.loc[:, donor_df.columns == miss_attr_list[k]]
                    arr_target = df_target.to_numpy()
                    svr_obj = SVR(kernel=kernel, C=C, epsilon=epsilon, gamma=gamma,
                    ↪  degree=degree, coef0=coef0)
                    reg = make_pipeline(StandardScaler(), svr_obj)
                    reg.fit(arr_feat, np.ravel(arr_target))
                    Pipeline(steps=[('standardscaler', StandardScaler()), ('svr',
                    ↪  svr_obj)])
                    svr_models_list.append(reg)
                svr_models.append(svr_models_list)
            return 0

        for i in range(len(recipient_df)):
            missing_attributes = []
            for j in range(cols_num):
                if recipient_df.iloc[i, j] == missing_value:
                    missing_attributes.append(recipient_df.columns[j])
            make_svr_models(missing_attributes)

            model_index = svr_missing_attributes.index(missing_attributes)
            recip_index = recipient_df.index[i]
            rec_df_target = recipient_df.loc[recip_index,
            ↪  ~recipient_df.columns.isin(missing_attributes)]
            rec_target = [rec_df_target.to_numpy().astype(float)]

            for m in range(len(missing_attributes)):
                prediction = svr_models[model_index][m].predict(rec_target)
                if attribute_types[missing_attributes[m]] == "float":
                    recipient_df.loc[recip_index, missing_attributes[m]] =
                    ↪  np.round(prediction[0] + 0.0001, decimals=3)
                else:
                    recipient_df.loc[recip_index, missing_attributes[m]] =
                    ↪  int(np.rint(prediction[0] + 0.0001))
            imp_record = recipient_df.loc[recip_index, :]
            df.loc[recip_index, :] = imp_record

        df.to_csv(imp_csv, index=False)

    except Exception as e:
        error_csv = r'imputation_error.csv'
```

```
82          shutil.copyfile(error_csv, imp_csv)
83          return 1
84
85      return 0
```

3.3.4 naive_bayes_imputer.py

This module implements Naive Bayes imputation algorithm for categorical datasets. The model that has been used here is 'CategoricalNB' from 'sklearn.naive_bayes' library.

This algorithm has similar steps and logic to the regression and SVR imputation modules that were described in the two previous sections. The difference is that the 'make_models' function uses the 'CategoricalNB' model to build the models and make the predictions and the imputation of missing values. In order to appropriately use the 'CategoricalNB' model, first some data preprocessing should be made. This is achieved with the use of 'OrdinalEncoder' module from 'sklearn.preprocessing' library. This module encodes the categorical features to appropriate numerical values that will be used as input in the 'CategoricalNB' model.

After the imputation is complete, the numerical values are transformed back to categorical using the inverse transformation of 'OrdinalEncoder' module and the Data Frame is stored in CSV format.

The source code of the module is shown below:

```
1   import shutil
2
3   import numpy as np
4   import pandas as pd
5   from sklearn.naive_bayes import CategoricalNB
6   from sklearn.preprocessing import OrdinalEncoder
7
8   from dataset_validator import is_categorical_attribute_types_valid
9
10  pd.options.mode.chained_assignment = None
11
12
13  def naive_bayes_imputer(filename, missing_value):
14      imp_csv = filename[:-4] + "_imp.csv"
15
16      try:
17          df = pd.read_csv(filename, low_memory=False)
18          attribute_types = df.loc[0, :]
19
20          if not is_categorical_attribute_types_valid(attribute_types, imp_csv):
21              return 1
22
23          cols_num = len(df.columns)
24          df = df.drop([0])
```

```
        df_enc = df.copy()
        ord_enc = OrdinalEncoder()
        df_enc[df_enc.columns] = ord_enc.fit_transform(df_enc[df_enc.columns])

        recipient_df = pd.DataFrame()
        recipient_df = df[(df == missing_value).any(axis=1)]

        donor_df = pd.DataFrame()
        donor_df = df.drop(recipient_df.index)

        donor_df_enc = donor_df.copy()

        donor_df_enc[donor_df_enc.columns] =
        ↪  ord_enc.transform(donor_df_enc[donor_df_enc.columns])

        recipient_df_enc = recipient_df.copy()
        recipient_df_enc[recipient_df_enc.columns] =
        ↪  ord_enc.transform(recipient_df_enc[recipient_df_enc.columns])

        nb_missing_attributes = []
        nb_models = []

        def make_nb_models(miss_attr_list: list):
            if miss_attr_list in nb_missing_attributes:
                return 0
            else:
                nb_missing_attributes.append(miss_attr_list)
                nb_models_list = []
                for k in range(len(miss_attr_list)):
                    df_enc_feat = donor_df_enc.loc[:, donor_df_enc.columns !=
                    ↪  miss_attr_list[k]]
                    df_enc_target = donor_df_enc.loc[:, donor_df_enc.columns ==
                    ↪  miss_attr_list[k]]
                    arr_enc_feat = df_enc_feat.to_numpy()
                    arr_enc_target = df_enc_target.to_numpy()
                    clf = CategoricalNB()
                    clf.fit(arr_enc_feat, np.ravel(arr_enc_target))
                    nb_models_list.append(clf)
                nb_models.append(nb_models_list)
            return 0

        for i in range(len(recipient_df_enc)):
            missing_attributes = []
            for j in range(cols_num):
                if recipient_df.iloc[i, j] == missing_value:
                    missing_attributes.append(recipient_df_enc.columns[j])

            make_nb_models(missing_attributes)
            model_index = nb_missing_attributes.index(missing_attributes)
```

```
            recip_index = recipient_df_enc.index[i]

            for m in range(len(missing_attributes)):
                rec_df_enc_target = recipient_df_enc.iloc[i,
                ↪  :].drop(labels=[missing_attributes[m]]).to_numpy()
                rec_target = [rec_df_enc_target]
                prediction = nb_models[model_index][m].predict(rec_target)
                recipient_df_enc.loc[recip_index, missing_attributes[m]] = prediction
            enc_record = [recipient_df_enc.loc[recip_index, :]]
            inv_record = ord_enc.inverse_transform(enc_record)
            df.loc[recip_index, :] = inv_record

        df.to_csv(imp_csv, index=False)

    except:
        error_csv = r'imputation_error.csv'
        shutil.copyfile(error_csv, imp_csv)
        return 1

    return 0
```

3.3.5 random_hot_deck_imputer.py

This module implements the Random Hot-Deck imputation algorithm.

The imputation algorithm is described as follows:

First the data from the CSV file are converted to a pandas Data Frame and a validation is made in order to ensure that the dataset has the correct form, meaning that numerical features should be declared under their name as 'int' for integer features, 'float' for floating point features and 'str' for categorical/string features.

Next the Data Frame is split into two Data Frames, one called the 'donor' and the other one called the 'recipient'. The 'donor' Data Frame contains all the records of the dataset that do not have any missing values. The 'recipient' Data Frame, on the other hand, contains all the records of the dataset that have at least one missing value.

After this separation the 'recipient' Data Frame is iterated and each of its records gets imputed by getting the corresponding values of a random record in the 'donor' Data Frame.

Finally the imputed Data Frame is stored in CSV format.

The source code of the module is shown below:

```
import shutil

import pandas as pd

```

```
from dataset_validator import is_mixed_attribute_types_valid

pd.options.mode.chained_assignment = None


def random_hot_deck_imputer(filename, missing_value):
    imp_csv = filename[:-4] + "_imp.csv"

    try:
        df = pd.read_csv(filename, low_memory=False)
        attribute_types = df.loc[0, :]

        if not is_mixed_attribute_types_valid(attribute_types, imp_csv):
            return 1

        df = df.drop([0])

        recipient_df = pd.DataFrame()
        recipient_df = df[(df == missing_value).any(axis=1)]

        donor_df = pd.DataFrame()
        donor_df = df.drop(recipient_df.index)

        for i in range(len(recipient_df)):
            donor_random_record = donor_df.sample(n=1)

            for j in range(len(recipient_df.columns)):
                if recipient_df.iloc[i, j] == missing_value:
                    df.loc[recipient_df.index[i], recipient_df.columns[j]] =
                    ↪  donor_random_record.iloc[0, j]

        df.to_csv(imp_csv, index=False)

    except:
        error_csv = r'imputation_error.csv'
        shutil.copyfile(error_csv, imp_csv)
        return 1

    return 0
```

3.3.6 numerical_knn_imputer.py

This module implements the numerical k-Nearest Neighbors (k-NN) imputation algorithm with the use of 'KNNImputer' from 'sklearn.impute' library.

The imputation algorithm is described as follows:

First the data from the CSV file are converted to a pandas Data Frame and a validation is made in order

to ensure that the dataset has the correct form, as has been analyzed in the previous sections.

Then the given missing value is replaced by 'np.nan' (numpy's 'not-a-number' constant) to prepare the input data in suitable format for the 'KNNImputer' model.

Additionally some data preprocessing is made using 'MinMaxScaler' module from 'sklearn.preprocessing' library. This module normalizes the data in the range of [0, 1]. Normalization (also called feature scaling) is an import preprocessing step of the k-NN imputation algorithm. Features generally have different scales or units and this can have as a result some features to dominate in the distance calculation just because they have larger values and this could lead to biased results.

After that an object of the 'KNNImputer' is created with k-Neighbors value and the normalized data are given as input to its 'fit_transform' method. This method handles the imputation of missing values based on the euclidean distance and taking the mean values of the k-Nearest Neighbors as the imputed values.

Finally the normalized imputed Data Frame is transformed back to its initial feature units and it is stored in CSV format.

The source code of the module is shown below:

```
import logging
import shutil

import numpy as np
import pandas as pd
from sklearn.impute import KNNImputer
from sklearn.preprocessing import MinMaxScaler

from dataset_validator import is_numerical_attribute_types_valid

import time

def numerical_knn_imputer(filename, missing_value, k_neighbors: int):
    start_time = time.time()

    imp_csv = filename[:-4] + "_imp.csv"

    try:
        df = pd.read_csv(filename, low_memory=False)
        attribute_types = df.loc[0, :]

        if not is_numerical_attribute_types_valid(attribute_types, imp_csv):
            return 1

        df = df.replace(to_replace=missing_value, value=np.nan)

        df_norm = df.copy()
        df_norm = df_norm.drop([0])
```

```
29
30          scaler = MinMaxScaler(feature_range=(0, 1))
31          df_norm = pd.DataFrame(scaler.fit_transform(df_norm), columns=df_norm.columns)
32
33          knn_imputer = KNNImputer(n_neighbors=k_neighbors)
34          df_norm_imputed = pd.DataFrame(knn_imputer.fit_transform(df_norm),
            ↪  columns=df_norm.columns)
35          df_imputed = pd.DataFrame(scaler.inverse_transform(df_norm_imputed),
            ↪  columns=df_norm.columns).round(3)
36
37          i = 0
38          for column in df_imputed.columns:
39              if attribute_types[i] == 'int':
40                  df_imputed[column] = df_imputed[column].astype(int)
41              i += 1
42
43          df_imputed.to_csv(imp_csv, index=False)
44
45          end_time = time.time()
46          duration=int((end_time-start_time)*1000)
47          print(f"duration(ms) = {duration}")
48      except Exception as e:
49          logging.error(e)
50          error_csv = r'imputation_error.csv'
51          shutil.copyfile(error_csv, imp_csv)
52          return 1
53
54      return 0
```

3.3.7 categorical_knn_imputer.py

This module implements the categorical k-Nearest Neighbors (k-NN) imputation algorithm. Since there was no relevant module in 'sklearn.impute' library, the algorithm was created from scratch, following the definition and the description of the k-Nearest Neighbors algorithm. The distance that has been used here is the 'hamming' distance.

The steps of this imputation algorithm are the following:

1. The data from the CSV file are converted to a pandas Data Frame and a validation is made in order to ensure that the dataset has the correct form, as has been analyzed in the previous sections.

2. The Data Frame is split to 'recipient' (contains at least one missing value) and 'donor' (no missing values).

3. For each record of the 'recipient' Data Frame the manhattan distances between the values of current non-missing feature values and the corresponding 'donor' values is calculated. These distances are saved as Data Frame, whose indices correspond to the indices of the 'donor' Data Frame. This is

very important, because the algorithm will later need to know which 'donor' records to use in order to impute the missing values of the 'recipient' record.

4. The sum of these distances is calculated and saved as Data Frame as well, keeping the same indices.
5. The Data Frame that contains the sum of the distances is sorted in ascending order along with its indices.
6. The first k indices of this sorted Data Frame are used to create a new Data Frame (the 'k_neighbors_df') that contains the k records of the 'donor' Data Frame which have the smallest distance to the current record of the 'recipient' Data Frame.
7. From the 'k_neighbors_df' the mode (most frequent) value of each feature is used to impute the current record of the 'recipient' Data Frame.
8. This procedure is repeated for all the records of the 'recipient' Data Frame.
9. The imputed Data Frame is stored in CSV format.

The source code of the module is shown below:

```
import shutil
from statistics import mean, median, mode

import numpy as np
import pandas as pd

from dataset_validator import is_categorical_attribute_types_valid


def categorical_knn_imputer(
        filename,
        missing_value,
        k_neighbors: int,
):
    imp_csv = filename[:-4] + "_imp.csv"

    try:
        df = pd.read_csv(filename, low_memory=False)
        attribute_types = df.loc[0, :]

        if not is_categorical_attribute_types_valid(attribute_types, imp_csv):
            return 1

        number_of_attributes = len(attribute_types)
        df = df.drop([0])

        # Split dataset to recipient (contains at least one missing value) and donor
        ↪  (no missing values)
```

```
        recipient_df = pd.DataFrame()
        recipient_df = df[(df == missing_value).any(axis=1)]
        donor_df = pd.DataFrame()
        donor_df = df.drop(recipient_df.index)

        # Calculate difference between two feature values
        def difference(x1, x2):
            return 0 if x1 == x2 else 1

        # Impute every row that contains missing value(s)
        for i in range(len(recipient_df)):
            # Get info and data for missing and present features for the current
            ↪ recipient_df record
            missing_record_index = recipient_df.index[i]
            missing_features_col_numbers = np.where(recipient_df.iloc[i, :] ==
            ↪ missing_value)[0]
            present_features_col_numbers = np.where(recipient_df.iloc[i, :] !=
            ↪ missing_value)[0]
            present_features_for_missing_record =
            ↪ recipient_df.loc[missing_record_index].iloc[present_features_col_numbers]
            present_features_donor_df = donor_df.iloc[:, present_features_col_numbers]

            # Calculate difference between the values of current non-missing feature
            ↪ values and the corresponding donor values
            df_difference = pd.DataFrame(
                [[difference(x, y) for x, y in zip(row,
                ↪ present_features_for_missing_record)] for row in
                present_features_donor_df.values],
                columns=present_features_donor_df.columns,
                index=present_features_donor_df.index
            )

            # Calculate sum of differences (equivalent to distance) and sort in
            ↪ ascending order
            sum_of_diff_df = pd.DataFrame()
            sum_of_diff_df['SumOfDiff'] = df_difference.sum(axis=1)
            sum_of_diff_df = sum_of_diff_df.sort_values(by='SumOfDiff')

            # Select the k-number of the sorted sum_of_diff_df dataframe which
            ↪ contains k-nearest neighbors
            k_neighbors_df = pd.DataFrame(columns=df.columns)
            for k in range(k_neighbors):
                k_index = sum_of_diff_df.index[k]
                k_neighbors_df.loc[k_index] = df.loc[k_index]

            # Create a result record from k_neighbors_df (mean or median of columns
            ↪ for numerical data, mode for categorical)
            result_k_neighbors_df = pd.DataFrame(columns=df.columns, index=[0])
            for j in range(number_of_attributes):
```

```
                column_data = k_neighbors_df.iloc[:, j]
                result_k_neighbors_df.iloc[0, j] = mode(column_data)

            # Impute the missing value of the current recipient record
            for missing_features_col_number in missing_features_col_numbers:
                df.loc[missing_record_index].iloc[missing_features_col_number] =
                ↪  result_k_neighbors_df.iloc[0, missing_features_col_number]

        df.to_csv(imp_csv, index=False)

    except Exception as e:
        error_csv = r'imputation_error.csv'
        shutil.copyfile(error_csv, imp_csv)
        return 1

    return 0
```

3.3.8 mixed_knn_imputer.py

This module implements the mixed k-Nearest Neighbors (k-NN) imputation algorithm for datasets that contain both numerical and categorical features. Since there was no relevant module in 'sklearn.impute' library, the algorithm was created from scratch, following the definition and the description of the k-Nearest Neighbors algorithm. The distance that has been used for categorical features is the 'hamming' distance, while there is the option between 'euclidean' and 'manhattan' for the numerical features.

The steps of this imputation algorithm are similar to categorical k-NN. The main differences are the following:

- The numerical features are normalized according to the selected distance type. For 'euclidean' distance they are normalized using the L2 norm, while for 'mahattan' distance they are normalized using the L1 norm for consistency reasons.
- For the numerical features either the mean value or the median (depending on the input option) is selected from the 'k_neighbors_df' Data Frame.

The source code of the module is shown below:

```
import shutil
from statistics import mean, median, mode

import numpy as np
import pandas as pd

from dataset_validator import is_attribute_types_valid


```

```
def mixed_knn_imputer(
        filename,
        missing_value,
        k_neighbors: int,
        metric='euclidean',
        numerical_method='mean'
):
    imp_csv = filename[:-4] + "_imp.csv"

    try:
        df = pd.read_csv(filename, low_memory=False)
        attribute_types = df.loc[0, :]

        if not is_attribute_types_valid(attribute_types, imp_csv):
            return 1

        number_of_attributes = len(attribute_types)

        # Normalize the numerical values according to metric (L2 normalization for
        ↪  euclidean, L1 for manhattan
        def normalize_array(array):
            num_values = []
            num_index = []
            for n in range(len(array)):
                if array[n] != missing_value:
                    num_values.append(float(array[n]))
                    num_index.append(n+1)
            num_array = np.array(num_values)
            if metric == "euclidean":
                normalized_array = num_array / np.linalg.norm(num_array)
            else:   # manhattan
                normalized_array = num_array / np.linalg.norm(num_array, ord=1)
            return num_index, normalized_array

        df = df.drop([0])
        norm_df = df.copy()

        for j in range(number_of_attributes):
            if attribute_types[j] != "str":
                normalized_obj = normalize_array(norm_df.iloc[:, j].values)
                for i in range(len(normalized_obj[0])):
                    norm_df.loc[normalized_obj[0][i], df.columns[j]] =
                    ↪  normalized_obj[1][i]

        # Split dataset to recipient (contains at least one missing value) and donor
        ↪  (no missing values)
        recipient_df = pd.DataFrame()
        recipient_df = norm_df[(df == missing_value).any(axis=1)]
        donor_df = pd.DataFrame()
```

```
56          donor_df = norm_df.drop(recipient_df.index)
57
58          # Calculate difference between two feature values according to attribute type
            ↪ and metric
59          def difference(x1, x2):
60              if type(x1) == str:
61                  return 0 if x1 == x2 else 1
62              elif metric == "euclidean":
63                  return (float(x1) - float(x2)) ** 2
64              else:  # manhattan
65                  return abs(float(x1) - float(x2))
66
67          # Impute every row that contains missing value(s)
68          for i in range(len(recipient_df)):
69              # Get info and data for missing and present features for the current
                ↪ recipient_df record
70              missing_record_index = recipient_df.index[i]
71              missing_features_col_numbers = np.where(recipient_df.iloc[i, :] ==
                ↪ missing_value)[0]
72              present_features_col_numbers = np.where(recipient_df.iloc[i, :] !=
                ↪ missing_value)[0]
73              present_features_for_missing_record =
                ↪ recipient_df.loc[missing_record_index].iloc[present_features_col_numbers]
74              present_features_donor_df = donor_df.iloc[:, present_features_col_numbers]
75
76              # Calculate difference between the values of current non-missing feature
                ↪ values and the corresponding donor values
77              df_difference = pd.DataFrame(
78                  [[difference(x, y) for x, y in zip(row,
                    ↪ present_features_for_missing_record)] for row in
79                   present_features_donor_df.values],
80                  columns=present_features_donor_df.columns,
81                  index=present_features_donor_df.index
82              )
83
84              # Calculate sum of differences (equivalent to distance) and sort in
                ↪ ascending order
85              sum_of_diff_df = pd.DataFrame()
86              sum_of_diff_df['SumOfDiff'] = df_difference.sum(axis=1)
87              sum_of_diff_df = sum_of_diff_df.sort_values(by='SumOfDiff')
88
89              # Select the k-number of the sorted sum_of_diff_df dataframe which
                ↪ contains k-nearest neighbors
90              k_neighbors_df = pd.DataFrame(columns=df.columns)
91              for k in range(k_neighbors):
92                  k_index = sum_of_diff_df.index[k]
93                  k_neighbors_df.loc[k_index] = df.loc[k_index]
94
```

```
95              # Create a result record from k_neighbors_df (mean or median of columns
                ↪  for numerical data, mode for categorical)
96              result_k_neighbors_df = pd.DataFrame(columns=df.columns, index=[0])
97              for j in range(number_of_attributes):
98                  column_data = k_neighbors_df.iloc[:, j]
99                  if attribute_types[j] == "str":
100                     result_k_neighbors_df.iloc[0, j] = mode(column_data)
101                 elif attribute_types[j] == "float":
102                     if numerical_method == "mean":
103                         result_k_neighbors_df.iloc[0, j] =
                            ↪  round(mean(column_data.astype(float)), 3)
104                     else:  # median
105                         result_k_neighbors_df.iloc[0, j] =
                            ↪  round(median(column_data.astype(float)), 3)
106                 else:  # int
107                     if numerical_method == "mean":
108                         result_k_neighbors_df.iloc[0, j] =
                            ↪  round(mean(column_data.astype(float)))
109                     else:  # median
110                         result_k_neighbors_df.iloc[0, j] =
                            ↪  round(median(column_data.astype(float)))
111
112             # Impute the missing value of the current recipient record
113             for missing_features_col_number in missing_features_col_numbers:
114                 df.loc[missing_record_index].iloc[missing_features_col_number] =
                    ↪  result_k_neighbors_df.iloc[0, missing_features_col_number]
115
116         df.to_csv(imp_csv, index=False)
117
118     except Exception as e:
119         error_csv = r'imputation_error.csv'
120         shutil.copyfile(error_csv, imp_csv)
121         return 1
122
123     return 0
```

3.3.9 dataset_validator.py

This module handles the validation of the dataset format. It checks if the attributes of the dataset have been defined according to the specifications of the application. Categorical features should be marked as 'str', meaning that the algorithm with handle them as strings. Numerical features should be marked either as 'int' in case they have integer values or 'float' in case they have floating point values.

If the validator finds an invalid feature type, the imputation procedure is terminated and the user receives a CSV with a corresponding error message.

For example, if a feature is marked as 'number', the validator will create a CSV with error message: "Invalid attribute type 'number'! Type must be either 'int' for integer attribute, 'float' for floating point

attribute or 'str' for string attribute!"

The source code of the module is shown below:

```
import csv

VALID_NUMERICAL_ATTRIBUTE_TYPES = [
    'int',
    'float'
]

VALID_CATEGORICAL_ATTRIBUTE_TYPES = [
    'str'
]

VALID_MIXED_ATTRIBUTE_TYPES = [
    'int',
    'float',
    'str'
]


def is_numerical_attribute_types_valid(attribute_types, imp_csv):
    for i in range(len(attribute_types)):
        if attribute_types[i] not in VALID_NUMERICAL_ATTRIBUTE_TYPES:
            error_message = [
                f"Invalid attribute type " + "\'" + attribute_types[i] + "\'" +
                "! Type must be either 'int' for integer attribute or 'float' for
                ↪  floating point attribute!"
            ]
            with open(imp_csv, mode='w', newline='') as csv_file:
                csv_writer = csv.writer(csv_file)
                csv_writer.writerow(error_message)
            return False
    return True


def is_categorical_attribute_types_valid(attribute_types, imp_csv):
    for i in range(len(attribute_types)):
        if attribute_types[i] not in VALID_CATEGORICAL_ATTRIBUTE_TYPES:
            error_message = [
                f"Invalid attribute type " + "\'" + attribute_types[i] + "\'" +
                "! Type must be 'str' for string/categorical attribute!"
            ]
            with open(imp_csv, mode='w', newline='') as csv_file:
                csv_writer = csv.writer(csv_file)
                csv_writer.writerow(error_message)
            return False
```

```
    return True


def is_mixed_attribute_types_valid(attribute_types, imp_csv):
    for i in range(len(attribute_types)):
        if attribute_types[i] not in VALID_MIXED_ATTRIBUTE_TYPES:
            error_message = [
                f"Invalid attribute type " + "\'" + attribute_types[i] + "\'" +
                "! Type must be either 'int' for integer attribute, 'float' for
                ↪  floating point attribute"
                "or 'str' for string/categorical attribute!"
            ]
            with open(imp_csv, mode='w', newline='') as csv_file:
                csv_writer = csv.writer(csv_file)
                csv_writer.writerow(error_message)
            return False
    return True
```

3.4 JavaScript Modules

In this section JavaScript modules that have been used in client-side are described and presented. All these modules are very useful because they perform actions and validations without overloading the server side.

3.4.1 prevent_upload.js

This module is applied on imputer.html page and is responsible for validating the CSV file extension and size before it gets uploaded to the server. This is very important because it prevents the upload of files with unsupported extensions (other than CSV) and large files that would overload the server resources. Each imputation method has its own maximum file size limit, as it requires different computation capacity. Additionally, this module performs validation on the float parameter fields (e.g. lasso regression alpha) , to ensure that correct values will be sent to the server.

The source code of the module is shown below:

```
alphaField = $("#alphaField");
l1RatioField = $("#l1RatioField");
svrGammaField = $("#svrGammaField");
svrCoef0Field = $("#svrCoef0Field");
svrCField = $("#svrCField");
svrEpsilonField = $("#svrEpsilonField");

let fileName = '';
let fileExtension = '';
let fileSize = 0;
let alphaValue = 1.0;
let l1RatioValue = 0.5;
let svrGammaValue = 0.3;
```

```
let svrCoef0Value = 0.0;
let svrCValue = 1.0;
let svrEpsilonValue = 0.1;

let file = document.getElementById("file")

$(document).ready(function() {
    file.value = "";
});

$('#file').bind('change', function() {
  fileName = this.files[0].name;
  fileExtension = fileName.split('.').pop().toLowerCase();
  fileSize = this.files[0].size;
});

alphaField.bind('change', function() {
  alphaValue = alphaField.val();
});

l1RatioField.bind('change', function() {
  l1RatioValue = l1RatioField.val();
});

svrGammaField.bind('change', function() {
  svrGammaValue = svrGammaField.val();
});

svrCoef0Field.bind('change', function() {
  svrCoef0Value = svrCoef0Field.val();
});

svrCField.bind('change', function() {
  svrCValue = svrCField.val();
});

svrEpsilonField.bind('change', function() {
  svrEpsilonValue = svrEpsilonField.val();
});

let maxFileSize = {
    "mean": [10485760, "10 MB"],
    "median": [10485760, "10 MB"],
    "mode": [10485760, "10 MB"],
    "mixed_mean_mode": [10485760, "10 MB"],
    "mixed_median_mode": [10485760, "10 MB"],
    "random_hot_deck": [10485760, "10 MB"],
    "linear_regression": [10485760, "10 MB"],
    "lasso_regression": [10485760, "10 MB"],
```

```
    "ridge_regression": [10485760, "10 MB"],
    "elastic_net_regression": [10485760, "10 MB"],
    "naive_bayes": [10485760, "10 MB"],
    "svr": [1048576, "1 MB"],
    "knn_mean": [5242880, "5 MB"],
    "knn_mode": [307200, "300 KB"],
    "mixed_knn_mean_mode": [307200, "300 KB"],
    "mixed_knn_median_mode": [307200, "300 KB"],
};

$("form").submit(function(e){
    let datasetFeaturesType = document.getElementById("datasetFeaturesType").value ;
    let impMethod = "none";
    if (datasetFeaturesType === "numerical"){
        impMethod = document.getElementById("numImpMethod").value ;
    }
    else if (datasetFeaturesType === "categorical"){
        impMethod = document.getElementById("catImpMethod").value ;
    }
    else{
        impMethod = document.getElementById("mixedImpMethod").value ;
    }

    if (fileExtension !== 'csv'){
        alert("Invalid file format! Only CSV files are allowed!");
        file.value = "";
        e.preventDefault(e);
    }

    if (fileSize > maxFileSize[impMethod][0]){
        alert("Dataset exceeds maximum allowed size for this method!
        ↪  ("+maxFileSize[impMethod][1]+")");
        file.value = "";
        e.preventDefault(e);
    }

    // First alphaValue is parsed as string in order to replace comma (if it exists)
    ↪  with dot.
    let alphaString = alphaValue.toString().replace(',', '.');
    let alphaFloat = parseFloat(alphaString);

    if (isNaN(alphaFloat) || alphaFloat < 0.001 || alphaFloat > 1000) {
        alert("alpha value must be a float number in range [0.001, 1000] due to
        ↪  computational limitations of the server!");
        e.preventDefault(e);
    } else {
        alphaField.val(alphaFloat);
    }

```

```
109     // First alphaValue is parsed as string in order to replace comma (if it exists)
        ↪ with dot.
110     let l1RatioString = l1RatioValue.toString().replace(',', '.');
111     let l1RatioFloat = parseFloat(l1RatioString);
112
113     if (isNaN(l1RatioFloat) || l1RatioFloat < 0 || l1RatioFloat > 1) {
114         alert("L1-ratio value must be a float number in range [0, 1]!");
115         e.preventDefault(e);
116     } else {
117         l1RatioField.val(l1RatioFloat);
118     }
119
120     // First svrGammaValue is parsed as string in order to replace comma (if it
        ↪ exists) with dot.
121     let svrGammaString = svrGammaValue.toString().replace(',', '.');
122     let svrGammaFloat = parseFloat(svrGammaString);
123
124     if (isNaN(svrGammaFloat) || svrGammaFloat < 0.001 || svrGammaFloat > 1000) {
125         alert("gamma value must be a float number in range [0.001, 1000] due to
            ↪ computational limitations of the server!");
126         e.preventDefault(e);
127     } else {
128         svrGammaField.val(svrGammaFloat);
129     }
130
131     // First svrCoef0Value is parsed as string in order to replace comma (if it
        ↪ exists) with dot.
132     let svrCoef0String = svrCoef0Value.toString().replace(',', '.');
133     let svrCoef0Float = parseFloat(svrCoef0String);
134
135     if (isNaN(svrCoef0Float) || svrCoef0Float < -10 || svrCoef0Float > 10) {
136         alert("coef0 value must be a float number in range [-10, 10] due to
            ↪ computational limitations of the server!");
137         e.preventDefault(e);
138     } else {
139         svrCoef0Field.val(svrCoef0Float);
140     }
141
142     // First svrCValue is parsed as string in order to replace comma (if it exists)
        ↪ with dot.
143     let svrCString = svrCValue.toString().replace(',', '.');
144     let svrCFloat = parseFloat(svrCString);
145
146     if (isNaN(svrCFloat) || svrCFloat < 0.001 || svrCFloat > 1000) {
147         alert("C value must be a float number in range [0.001, 1000] due to
            ↪ computational limitations of the server!");
148         // document.getElementById("file").value = "";
149         e.preventDefault(e);
150     } else {
```

```
151             svrCField.val(svrCFloat);
152         }
153 
154         // First svrEpsilonValue is parsed as string in order to replace comma (if it
            ↪  exists) with dot.
155         let svrEpsilonString = svrEpsilonValue.toString().replace(',', '.');
156         let svrEpsilonFloat = parseFloat(svrEpsilonString);
157 
158         if (isNaN(svrEpsilonFloat) || svrEpsilonFloat < 0.1 || svrEpsilonFloat > 1000) {
159             alert("epsilon value must be a float number in range [0.1, 1000] due to
                ↪  computational limitations of the server!");
160             e.preventDefault(e);
161         } else {
162             svrEpsilonField.val(svrEpsilonFloat);
163         }
164 });
```

3.4.2 show_available_methods_and_parameters.js

This module is also applied on imputer.html page and handles the filtering of the available imputation methods according to the selected dataset type, as described in section 3.2.

It also handles the filtering of method parameters, such as 'alpha' for lasso regression or 'k-Neighbors' for k-NN. These parameters are only displayed when the corresponding imputation method is selected, otherwise they get hidden, in order to improve the user experience.

The source code of the module is shown below:

```
1  let datasetFeaturesType = document.getElementById("datasetFeaturesType");
2  let numImpMethod = document.getElementById("numImpMethod");
3  let catImpMethod = document.getElementById("catImpMethod");
4  let mixedImpMethod = document.getElementById("mixedImpMethod");
5  let svrKernelOptions = document.getElementById("svrKernelOptions");
6  let svrGammaOptions = document.getElementById("svrGammaOptions");
7 
8  let knn_neighbors = $('#knn_neighbors');
9  let knn_neighbors_options = $('#knn_neighbors_options');
10 let knn_distance = $('#knn_distance');
11 let knn_distance_options = $('#knn_distance_options');
12 let alpha = $('#alpha');
13 let alpha_input = $('#alpha_input');
14 let l1_ratio = $('#l1_ratio');
15 let l1_ratio_input = $('#l1_ratio_input');
16 let svr_kerner = $('#svr_kernel');
17 let svr_kerner_options = $('#svr_kernel_options');
18 let svr_poly_degree = $('#svr_poly_degree');
19 let svr_poly_degree_options = $('#svr_poly_degree_options');
20 let svr_coef0 = $('#svr_coef0');
```

```
let svr_coef0_input = $('#svr_coef0_input');
let svr_c = $('#svr_c');
let svr_c_input = $('#svr_c_input');
let svr_epsilon = $('#svr_epsilon');
let svr_epsilon_input = $('#svr_epsilon_input');
let svr_gamma = $('#svr_gamma');
let svr_gamma_options = $('#svr_gamma_options');
let svr_gamma_input = $('#svr_gamma_input');

let num_imp_method = $('#num_imp_method');
let num_imp_method_options = $('#num_imp_method_options');
let cat_imp_method =$('#cat_imp_method');
let cat_imp_method_options = $('#cat_imp_method_options');
let mixed_imp_method = $('#mixed_imp_method');
let mixed_imp_method_options = $('#mixed_imp_method_options');

 datasetFeaturesType.onchange = function(){
     hide_knn();
     hide_alpha();
     hide_l1_ratio();
     hide_svr();

     let featuresType = datasetFeaturesType.value;

     if (featuresType === "categorical") {
          num_imp_method.hide();
          num_imp_method_options.hide();
          cat_imp_method.show();
          cat_imp_method_options.show();
          mixed_imp_method.hide();
          mixed_imp_method_options.hide();
          if (catImpMethod.value === "knn_mode"){
              show_only_knn_neighbors();
              hide_alpha();
              hide_l1_ratio();
              hide_svr();
          }
    } else if (featuresType === "mixed"){
          num_imp_method.hide();
          num_imp_method_options.hide();
          cat_imp_method.hide();
          cat_imp_method_options.hide();
          mixed_imp_method.show();
          mixed_imp_method_options.show();
          if (mixedImpMethod.value === "mixed_knn_mean_mode" || mixedImpMethod.value
          ↪  === "mixed_knn_median_mode"){
              show_knn();
              hide_alpha();
              hide_l1_ratio();
```

```
69              hide_svr();
70          }
71      } else {
72          num_imp_method.show();
73          num_imp_method_options.show();
74          cat_imp_method.hide();
75          cat_imp_method_options.hide();
76          mixed_imp_method.hide();
77          mixed_imp_method_options.hide();
78          if (numImpMethod.value === "knn_mean"){
79              show_only_knn_neighbors();
80              hide_alpha();
81              hide_l1_ratio();
82              hide_svr();
83          } else if (numImpMethod.value === "lasso_regression" || numImpMethod.value
            ↪  === "ridge_regression") {
84              hide_knn();
85              show_alpha();
86              hide_l1_ratio();
87              hide_svr();
88          } else if (numImpMethod.value === "elastic_net_regression") {
89              hide_knn();
90              show_alpha();
91              show_l1_ratio();
92              hide_svr();
93          } else if (numImpMethod.value === "svr") {
94              hide_knn();
95              hide_alpha();
96              hide_l1_ratio();
97              show_svr();
98          }
99      }
100 }
101
102 numImpMethod.onchange = function() {
103     let num_imp_method = numImpMethod.value;
104
105     if (num_imp_method === "knn_mean") {
106         show_only_knn_neighbors();
107         hide_alpha();
108         hide_l1_ratio();
109         hide_svr();
110     } else if (numImpMethod.value === "lasso_regression" || numImpMethod.value ===
        ↪  "ridge_regression") {
111         hide_knn();
112         show_alpha();
113         hide_l1_ratio();
114         hide_svr();
115     } else if (numImpMethod.value === "elastic_net_regression") {
```

```
116             hide_knn();
117             show_alpha();
118             show_l1_ratio();
119             hide_svr();
120         } else if (num_imp_method === "svr") {
121             hide_knn();
122             hide_alpha();
123             hide_l1_ratio();
124             show_svr();
125         } else {
126             hide_knn();
127             hide_alpha();
128             hide_l1_ratio();
129             hide_svr();
130        }
131     }
132
133     catImpMethod.onchange = function() {
134         let cat_imp_method = catImpMethod.value;
135
136         if (cat_imp_method === "knn_mode") {
137             hide_alpha();
138             hide_l1_ratio();
139             show_only_knn_neighbors();
140             hide_svr();
141         } else {
142             hide_alpha();
143             hide_l1_ratio();
144             hide_knn();
145             hide_svr();
146         }
147     }
148
149     mixedImpMethod.onchange = function() {
150         let mixed_imp_method = mixedImpMethod.value;
151
152         if (mixed_imp_method === "mixed_knn_mean_mode" || mixed_imp_method ===
            ↪  "mixed_knn_median_mode") {
153             hide_alpha();
154             hide_l1_ratio();
155             show_knn();
156             hide_svr();
157         } else {
158             hide_alpha();
159             hide_l1_ratio();
160             hide_knn();
161             hide_svr();
162         }
163     }
```

```
164
165  function hide_knn() {
166      knn_neighbors.hide();
167      knn_neighbors_options.hide();
168      knn_distance.hide();
169      knn_distance_options.hide();
170  }
171
172  function show_knn() {
173      knn_neighbors.show();
174      knn_neighbors_options.show();
175      knn_distance.show();
176      knn_distance_options.show();
177  }
178
179  function show_only_knn_neighbors() {
180      knn_neighbors.show();
181      knn_neighbors_options.show();
182      knn_distance.hide();
183      knn_distance_options.hide();
184  }
185
186  function hide_alpha() {
187      alpha.hide();
188      alpha_input.hide();
189  }
190
191 function show_alpha() {
192      alpha.show();
193      alpha_input.show();
194 }
195
196 function hide_l1_ratio() {
197      l1_ratio.hide();
198      l1_ratio_input.hide();
199 }
200
201 function show_l1_ratio() {
202      l1_ratio.show();
203      l1_ratio_input.show();
204 }
205
206 function hide_svr() {
207      svr_kerner.hide();
208      svr_kerner_options.hide();
209      svr_gamma.hide();
210      svr_gamma_options.hide();
211      svr_gamma_input.hide();
212      svr_poly_degree.hide();
```

```
213       svr_poly_degree_options.hide();
214       svr_coef0.hide();
215       svr_coef0_input.hide();
216       svr_c.hide();
217       svr_c_input.hide();
218       svr_epsilon.hide();
219       svr_epsilon_input.hide();
220   }
221
222   svrKernelOptions.onchange = function() {
223       show_svr();
224   }
225
226   svrGammaOptions.onchange = function() {
227       show_or_hide_svr_gamma_input();
228   }
229
230   function show_svr() {
231       let svrKernel = svrKernelOptions.value;
232
233       if (svrKernel === "rbf") {
234           show_svr_rbf();
235       } else if (svrKernel === "linear") {
236           show_svr_linear();
237       } else if (svrKernel === "poly") {
238           show_svr_poly();
239       } else {
240           show_svr_sigmoid();
241       }
242   }
243
244   function show_svr_rbf() {
245       svr_kerner.show();
246       svr_kerner_options.show();
247       show_or_hide_gamma();
248       show_or_hide_svr_gamma_input();
249       svr_poly_degree.hide();
250       svr_poly_degree_options.hide();
251       svr_coef0.hide();
252       svr_coef0_input.hide();
253       svr_c.show();
254       svr_c_input.show();
255       svr_epsilon.show();
256       svr_epsilon_input.show();
257   }
258
259   function show_or_hide_gamma() {
260       if (svrKernelOptions.value === "rbf") {
261           svr_gamma.show();
```

```
262             svr_gamma_options.show();
263             show_or_hide_svr_gamma_input();
264         } else {
265             svr_gamma.hide();
266             svr_gamma_options.hide();
267             svr_gamma_input.hide();
268         }
269     }
270
271     function show_or_hide_svr_gamma_input() {
272         if (svrGammaOptions.value === "custom") {
273             svr_gamma_input.show();
274         } else {
275             svr_gamma_input.hide();
276         }
277     }
278     function show_svr_linear() {
279         svr_kerner.show();
280         svr_kerner_options.show();
281         svr_gamma.hide();
282         svr_gamma_options.hide();
283         svr_gamma_input.hide();
284         svr_poly_degree.hide();
285         svr_poly_degree_options.hide();
286         svr_coef0.hide();
287         svr_coef0_input.hide();
288         svr_c.show();
289         svr_c_input.show();
290         svr_epsilon.show();
291         svr_epsilon_input.show();
292     }
293
294     function show_svr_poly() {
295         svr_kerner.show();
296         svr_kerner_options.show();
297         svr_gamma.hide();
298         svr_gamma_options.hide();
299         svr_gamma_input.hide();
300         svr_poly_degree.show();
301         svr_poly_degree_options.show();
302         svr_coef0.show();
303         svr_coef0_input.show();
304         svr_c.show();
305         svr_c_input.show();
306         svr_epsilon.show();
307         svr_epsilon_input.show();
308     }
309
310     function show_svr_sigmoid() {
```

```
311         svr_kerner.show();
312         svr_kerner_options.show();
313         svr_gamma.hide();
314         svr_gamma_options.hide();
315         svr_gamma_input.hide();
316         svr_poly_degree.hide();
317         svr_poly_degree_options.hide();
318         svr_coef0.show();
319         svr_coef0_input.show();
320         svr_c.show();
321         svr_c_input.show();
322         svr_epsilon.show();
323         svr_epsilon_input.show();
324 }
```

3.4.3 show_download_link.js

This module is applied on success.html page and is responsible for showing the download link of the imputed dataset, as soon as the imputation process finishes.

The whole process is as follows:

1. First a loading gif and a message appear, showing that the imputation process is still in progress
2. Then it proceeds to make a HEAD call to the server, trying to fetch the imputed dataset. The HEAD method has been selected as it is faster than GET and it is more suitable for this case, because only the status code of the response is required to check if the imputed file exists on the server.
3. The HEAD call is repeated five times in total. The interval between the calls has been set to five seconds, in order not to overload the server.
4. If the status code of the response to one of the HEAD calls is 200 (meaning that the imputed file exists on the server) the loading gif and the corresponding message get hidden and the download link appears. The user then can click on the link to download the imputed dataset.
5. If the status code of all responses to the HEAD calls is 404 (meaning the file does not exist on the server) or any other status code (meaning that there is some other problem on the server-side) a message appears showing that the imputation progress requires more time. Also, along with that the download link of the imputed dataset appears suggesting to the user to use it after a few minutes. This case usually applies when the dataset size is close to the maximum allowed file size.

The source code of the module is shown below:

```
1 function ImpFileExists() {
2     let imp_file_url = document.getElementById("download_link").href;
3     let http = new XMLHttpRequest();
```

```
    let timeout_cnt = 0;
    let timeout_cnt_max = 3;
    let sleepTimeInMilliseconds = 5000;

    function sleep(ms) {
        return new Promise(resolve => setTimeout(resolve, ms));
    }

    async function sendRequestAndWait(){
        while (sendRequest() === false && timeout_cnt <= timeout_cnt_max){
            await sleep(sleepTimeInMilliseconds);
            timeout_cnt += 1;
        }
        if (timeout_cnt > timeout_cnt_max){
            $('#loading_gif').hide();
            $('#please_wait').hide();
            $('#imputation_completed').hide();
            $('#download_button').hide();
            $('#more_time').show();
            $('#download_button_2').show();
        }
    }

    sendRequestAndWait();

    function sendRequest() {
        http.open('HEAD', imp_file_url, false);
        http.send();
        if (http.status === 200) {
            $('#loading_gif').hide();
            $('#please_wait').hide();
            $('#imputation_completed').show();
            $('#download_button').show();
            return true
        }
        else{
            return false
        }
    }
}
```

Chapter 4: WEBIMPUTER Presentation

4.1 Home Page

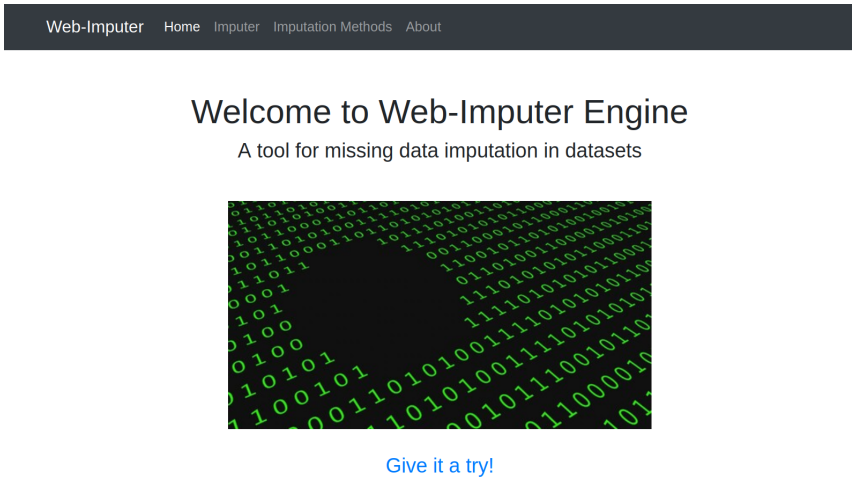

Figure 4.1: Home Page

Figure 4.1 displays the home page of the application. The navigation bar consists of WEBIMPUTER logo, which has a link to home page, the 'Home' button, the 'Imputer' button which has a link to the Imputer Page, the 'Imputation Methods' button which has a link to imputation methods page and the 'About' button which has a link to the 'About' page.

4.2 Imputer Page

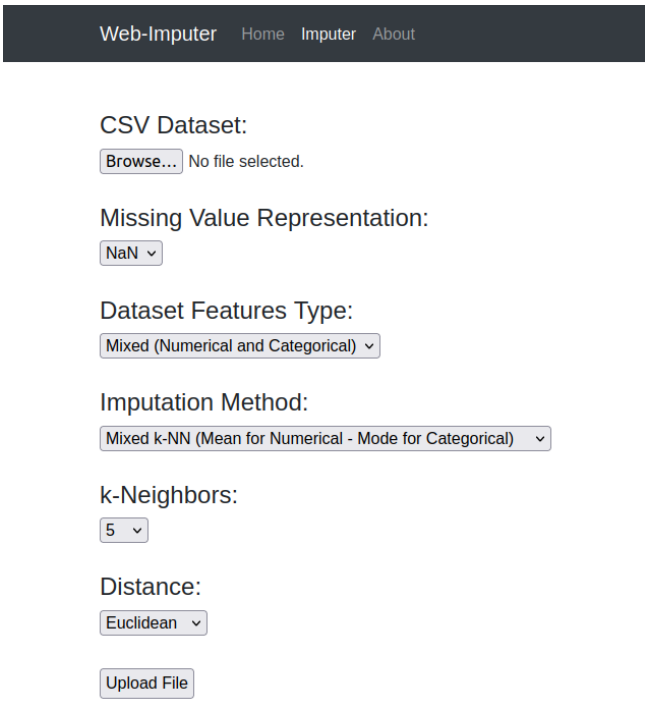

Figure 4.2: Imputer Page

Figure 4.2 displays the imputer page. On this page, the user uploads the dataset with missing values and fills in all the necessary information (’Missing Value Represantation’, ’Dataset Features Type’, ’Imputation Method’ and perhaps some additional parameters that are required for some algorithms, such as k-NN).

Then he/she submits the form and waits for the response which will contain the download link for the imputed dataset in CSV format.

Here some validations are made before the CSV gets uploaded to the server and the user might receive some error messages, if the CSV and the chosen parameters are not according to the required specifications of the application.

4.3 Imputation Methods Page

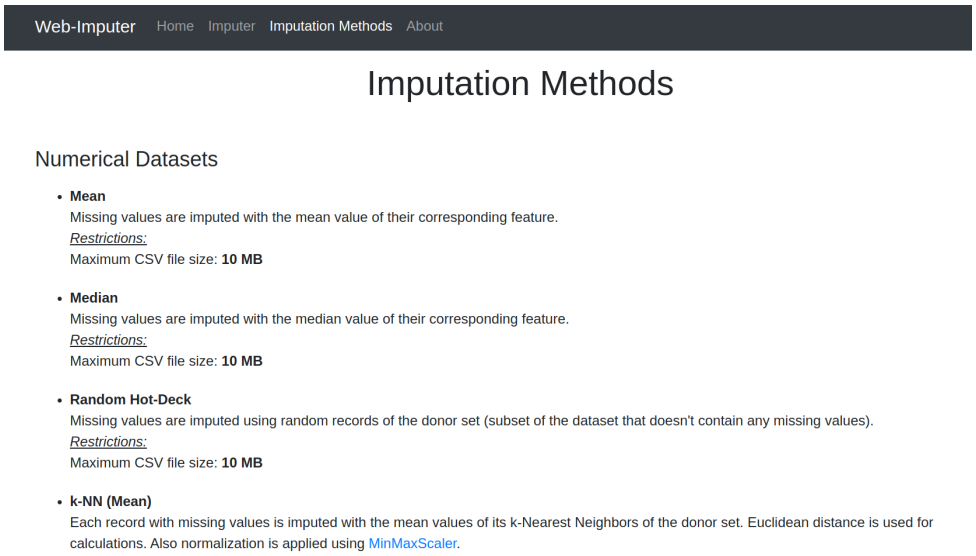

Figure 4.3: Imputation Methods Page

Figure 4.3 displays the imputation methods page. On this page a short presentation and description of each imputation methods is made, along with some restrictions (allowed range of parameters and maximum allowed CSV file size).

4.4 About Page

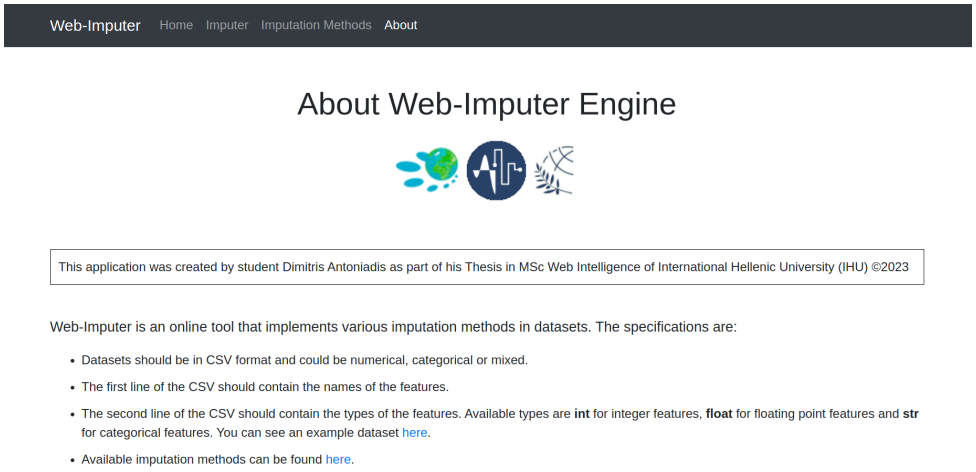

Figure 4.4: About Page

Figure 4.4 displays the about page. On this page the user gets some general information about the application and the format and of the CSV file that should be uploaded, as well as a link to the available imputations methods page.

4.5 Success Response

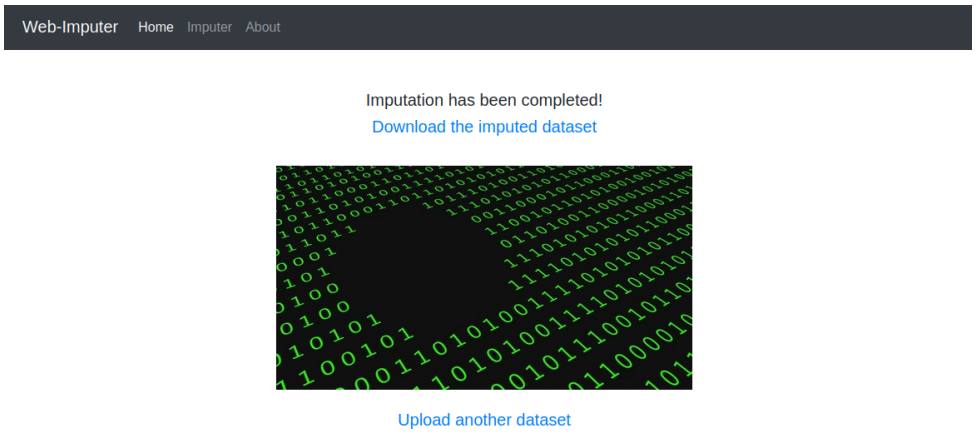

Figure 4.5: Success Response: Imputation completed

Figure 4.5 displays the success page. This page is rendered and returned from the server when the imputation progress is completed. It includes a download link to the imputed dataset in CSV format. The name of the file is the same with the one that the user uploaded with ”_imp” as suffix. For example if the user uploaded ”cars.csv” the name of the imputed file will be ”cars_imp.csv”.

4.6 Validation and Errors

In this section some typical validations that are made in the front-end using the JavaScript modules are presented. In addition, some other HTML pages that are rendered and returned from the server after some

checks are also dispayed here.

4.6.1 Invalid File Format

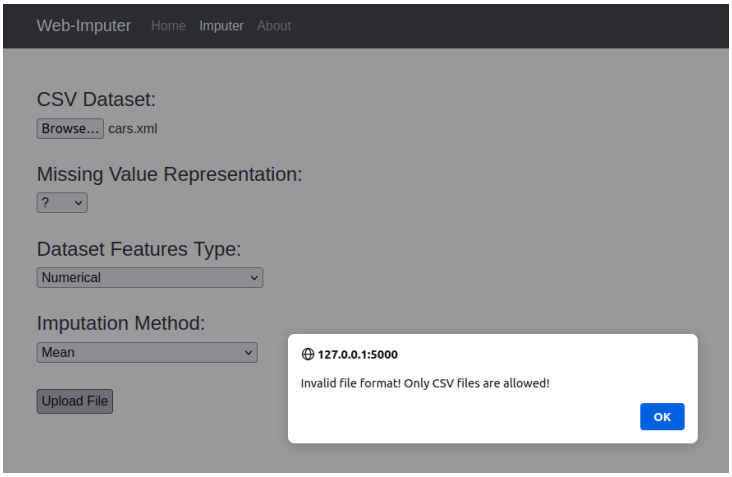

Figure 4.6: Upload validation: Invalid file format

Figure 4.6 displays the pop-up error message that the user receives after trying to upload a dataset file with extension other than CSV. This check is performed in the client-side, in order not to use unnecessarily the serve's resources.

4.6.2 Invalid Model Parameters

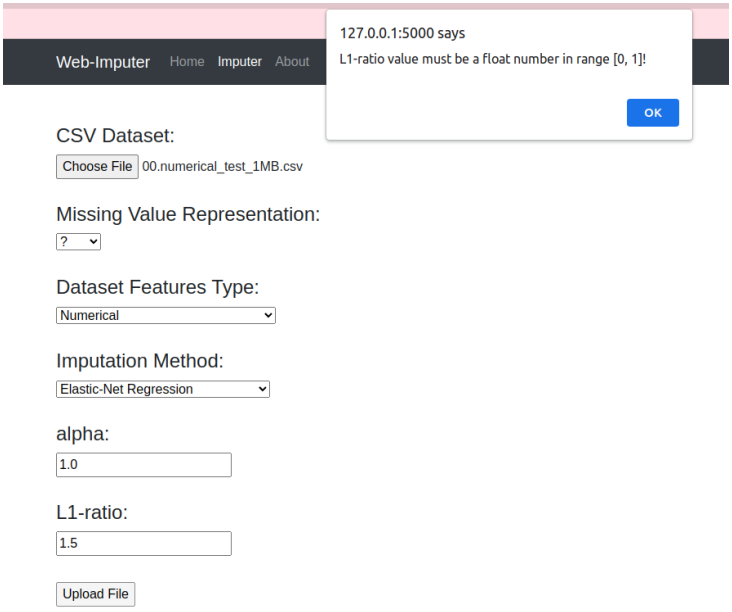

Figure 4.7: Upload validation: Invalid model parameters

Figure 4.7 displays the pop-up error message that the user receives after inserting invalid or out of allowed range model parameters. This error message is only relevant to the input fields that user writes a custom number. So if for example the user writes a text instead of a number or inserts a very large number that

is not acceptable, this message will appear. Such validation does not apply to drop-down type fields, as the available choices are pre-determined by creator of the application.

4.6.3 Dataset Exceeds Maximum Size

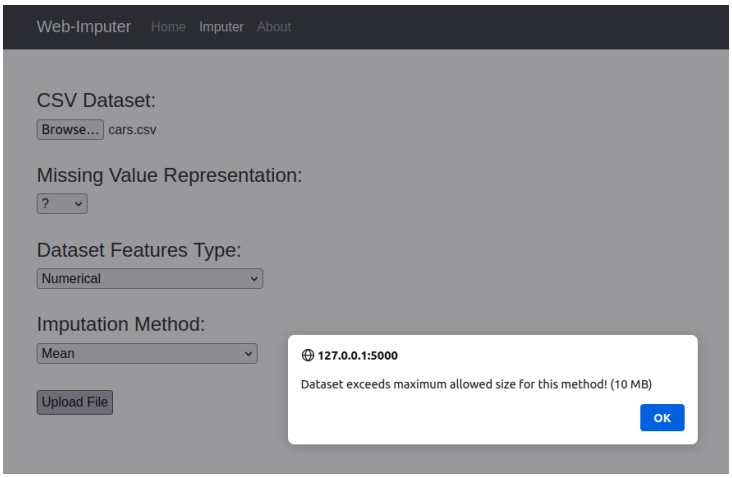

Figure 4.8: Upload validation: Dataset exceeds maximum allowed size

Figure 4.8 displays the pop-up error message that the user receives after trying to upload a dataset file with size larger than the maximum allowed. This check is performed in the client-side, in order not to overload the server.

Otherwise if the file would first be uploaded to the server and then the validation was made, the server would have to handle the upload of very large files which could lead to decreased performance and security.

Each imputation method has its own maximum allowed file size limit. This happens due to the fact that some methods are more complex than the others and require a lot of computational capacity, as they have to perform millions of calculations. Thus the more complex an imputation method is the lower maximum allowed file size limit has been set.

These limits have been set according to experiments with various methods, parameters and file sizes that have been made. The general strategy that has been applied is that the maximum completion time should not exceed the three-minute threshold in any case, in order not to exceed the server's computational capacity.

Below is a list of the available methods and their corresponding maximum allowed file size limits:

- Mean: **10 MB**
- Median: **10 MB**
- Mode: **10 MB**
- Mixed (Mean for Numerical - Mode for Categorical): **10 MB**

- Mixed (Median for Numerical - Mode for Categorical): **10 MB**
- Random Hot-Deck: **10 MB**
- Linear Regression: **10 MB**
- Lasso Regression: **10 MB**
- Ridge Regression: **10 MB**
- Elastic-Net Regression: **10 MB**
- Naive-Bayes: **10 MB**
- Support Vector Regression (SVR): **1 MB**
- Numerical k-NN (Mean): **5 MB**
- Categorical k-NN (Mode): **300 KB**
- Mixed k-NN (Mean for Numerical - Mode for Categorical): **300 KB**
- Mixed k-NN (Median for Numerical - Mode for Categorical): **300 KB**

4.6.4 Imputation Still in Progress

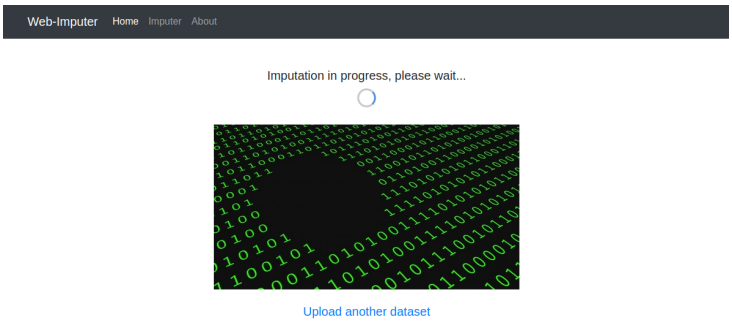

Figure 4.9: Imputation still in progress

Figure 4.9 displays the page the user sees when he/she is waiting for the imputation procedure to be completed. It has a loading gif and a message, indicating that the imputation is still in progress. After the imputation is completed, a download link to the imputed dataset appears (see success page).

In case the user has uploaded a large dataset and has selected a complex method that requires a lot of calculations and time (e.g. mixed k-NN), after about 20 seconds, he/she will see the message that writes ”Imputation requires more time...” (see next section).

4.6.5 Imputation Requires More Time

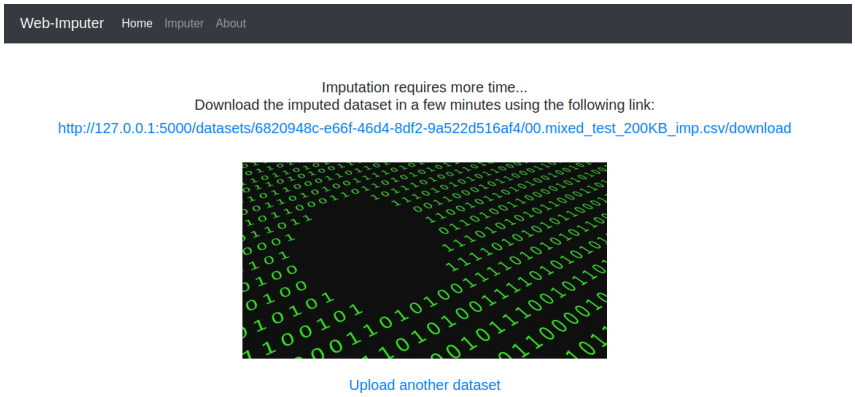

Figure 4.10: Imputation requires more time

Figure 4.10 displays the page that the user sees when the imputation process takes more than 20 seconds. Usually this is the case when large datasets with file size close to maximum allowed limit are combined with the selection of complex imputation methods and/or model parameters that lead to increased computational resources and thus increased completion time.

Along with the message that writes ”Imputation requires more time...”, the download link is provided and the user is suggested to try use this link to download the imputed dataset in a few minutes or whenever he/she likes in the future.

Chapter 5: WEBIMPUTER Experiments and Metrics

In this section experiments and metrics for WEBIMPUTER are presented. More specifically, the available imputation methods of the application have been tested on datasets with a typical percentage of 6.5% missing values and for various file sizes. The corresponding execution time has been monitored and is displayed here.

The technical specifications of the machine that has been used for these experiments are:

- **CPU:** AMD Ryzen 7 5825U (min 1600 MHz, max 4547 MHz)
- **RAM:** 30 GB

The restrictions of each method described on the relevant sections (see 3.4.1 and 4.3) have been decided based on these experiments and metrics. The criteria of choosing the maximum allowed CSV file size were two:

- The file size should not be larger than 10 MB.
- The execution time should not be more than three minutes.

These restrictions are necessary in order not to overload the server.

5.1 Numerical Datasets

5.1.1 Mean

Table 5.1 displays the execution time of mean imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	0.2
5	0.7
10	1.3

Table 5.1: Execution time for mean

5.1.2 Median

Table 5.2 displays the execution time of median imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	0.2
5	0.7
10	1.4

Table 5.2: Execution time for median

5.1.3 Random Hot-Deck

Table 5.3 displays the execution time of random hot-deck imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	3.3
5	19.5
10	55

Table 5.3: Execution time for random hot-deck (numerical dataset)

5.1.4 k-NN (Mean)

Table 5.4 displays the execution time of k-NN (Mean) imputation method on various numerical datasets of different file size and for different values of k-Neighbors:

Dataset Size (MB)	k-Neighbors	Execution Time (sec)
1	3	6
	10	6
	20	6
2	3	23
	10	21
	20	21
3	3	53
	10	48
	20	48
5	3	88
	10	76
	20	76

Table 5.4: Execution time for k-NN (Mean)

5.1.5 Linear Regression

Table 5.5 displays the execution time of random hot-deck imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	5
5	20
10	41

Table 5.5: Execution time for linear regression

5.1.6 Lasso Regression

Table 5.6 displays the execution time of lasso regression imputation method on various numerical datasets of different file size and for different values of alpha (α):

Dataset Size (MB)	alpha	Execution Time (sec)
1	0.001	6
	1	8
	1000	6
5	0.001	22
	1	24
	1000	27
10	0.001	42
	1	43
	1000	47

Table 5.6: Execution time for lasso regression

5.1.7 Ridge Regression

Table 5.7 displays the execution time of ridge regression imputation method on various numerical datasets of different file size and for different values of alpha (α):

Dataset Size (MB)	alpha	Execution Time (sec)
1	0.001	4
	1	4
	1000	4
5	0.001	19
	1	17
	1000	18
10	0.001	36
	1	35
	1000	35

Table 5.7: Execution time for ridge regression

5.1.8 Elastic-Net Regression

Table 5.8 displays the execution time of elastic-net regression imputation method on various numerical datasets of different file size and for different values of alpha (α) and *L1-ratio*:

Dataset Size (MB)	alpha	L1-ratio	Execution Time (sec)
1	0.001	0.25	5
		0.5	5
		0.75	5
	1	0.25	8
		0.5	8
		0.75	7
	1000	0.25	6
		0.5	5
		0.75	6
5	0.001	0.25	22
		0.5	22
		0.75	23
	1	0.25	22
		0.5	24
		0.75	22
	1000	0.25	43
		0.5	40
		0.75	38
10	0.001	0.25	44
		0.5	43
		0.75	42
	1	0.25	42
		0.5	42
		0.75	42
	1000	0.25	72
		0.5	78
		0.75	71

Table 5.8: Execution time for elastic-net regression

5.1.9 Support Vector Regression (SVR)

Table 5.9 displays the execution time of SVR imputation method with RBF kernel on a numerical dataset of 1 MB and for different values of gamma (γ), *C* and *epsilon* (ϵ):

gamma	C	epsilon	Execution Time (sec)
auto	0.001	0.1	38
		1	37
		10	29
	1	0.1	21
		1	10
		10	7
	1000	0.1	13
		1	6
		10	6
scale	0.001	0.1	38
		1	36
		10	28
	1	0.1	22
		1	10
		10	7
	1000	0.1	13
		1	6
		10	6
0.01	0.001	0.1	38
		1	37
		10	29
	1	0.1	28
		1	15
		10	8
	1000	0.1	15
		1	7
		10	6
0.1	0.001	0.1	39
		1	36
		10	29
	1	0.1	19
		1	10
		10	7
	1000	0.1	14
		1	7
		10	6
1	0.001	0.1	37
		1	37
		10	30
	1	0.1	21
		1	15
		10	11
	1000	0.1	19
		1	13
		10	10

Table 5.9: Execution time for SVR with RBF kernel (1 MB dataset)

Table 5.10 displays the execution time SVR imputation method with linear kernel on a numerical dataset of 1 MB and for different values of *C* and *epsilon* (ϵ):

C	epsilon	Execution Time (sec)
0.001	0.1	23
	1	20
	10	14
1	0.1	6
	1	6
	10	6
1000	0.1	6
	1	6
	10	6

Table 5.10: Execution time for SVR with linear kernel (1 MB dataset)

Table 5.11 displays the execution time of SVR imputation method with polynomial kernel of 3rd degree on a numerical dataset of 1 MB and for different values of *coef0*, *C* and *epsilon* (ϵ):

coef0	C	epsilon	Execution Time (sec)
-10	0.001	0.1	35
		1	34
		10	23
	1	0.1	37
		1	37
		10	30
	1000	0.1	37
		1	38
		10	30
0	0.001	0.1	30
		1	29
		10	23
	1	0.1	39
		1	20
		10	10
	1000	0.1	42
		1	10
		10	6
10	0.001	0.1	15
		1	14
		10	10
	1	0.1	10
		1	10
		10	7
	1000	0.1	10
		1	9
		10	7

Table 5.11: Execution time for SVR with polynomial kernel of 3rd degree (1 MB dataset)

Table 5.12 displays the execution time of SVR imputation method with polynomial kernel of 5th degree on a numerical dataset of 1 MB and for different values of *coef0*, *C* and *epsilon* (ϵ):

coef0	C	epsilon	Execution Time (sec)
-10	0.001	0.1	37
			35
			33
	1	0.1	36
			38
			35
	1000	0.1	37
			39
			34
0	0.001	0.1	32
			32
			22
	1	0.1	60
			46
			14
	1000	0.1	157
			40
			7
10	0.001	0.1	10
			8
			7
	1	0.1	10
			8
			7
	1000	0.1	10
			8
			7

Table 5.12: Execution time for SVR with polynomial kernel of 5th degree (1 MB dataset)

Table 5.13 displays the execution time of SVR imputation method with sigmoid kernel on a numerical dataset of 1 MB and for different values of *coef0*, *C* and *epsilon* (ϵ):

coef0	C	epsilon	Execution Time (sec)
-10	0.001	0.1	54
		1	53
		10	40
	1	0.1	56
		1	53
		10	40
	1000	0.1	55
		1	52
		10	40
0	0.001	0.1	49
		1	50
		10	37
	1	0.1	49
		1	49
		10	50
	1000	0.1	49
		1	51
		10	46
10	0.001	0.1	51
		1	50
		10	40
	1	0.1	53
		1	51
		10	40
	1000	0.1	51
		1	50
		10	40

Table 5.13: Execution time for SVR with sigmoid kernel (1 MB dataset)

5.1.10 Methods Comparison

Figure 5.1 displays the average execution time for all numerical imputation methods applied to a typical 1 MB dataset with 6.5% missing values.

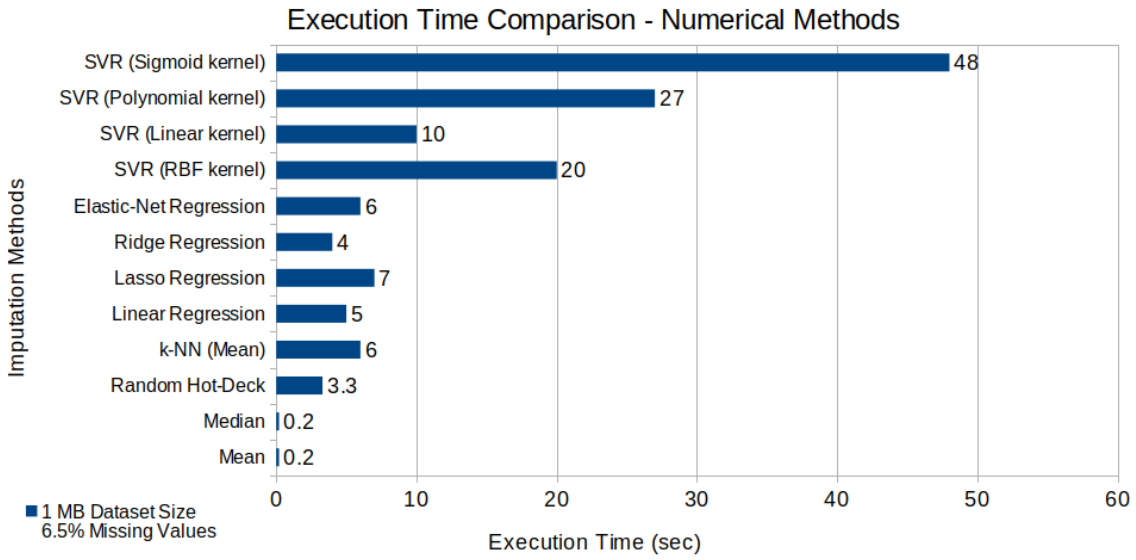

Figure 5.1: Execution Time Comparison - Numerical Methods

5.2 Categorical Datasets

5.2.1 Mode

Table 5.14 displays the execution time of Mode imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	0.1
5	0.2
10	0.4

Table 5.14: Execution time for mode

5.2.2 Random Hot-Deck

Table 5.15 displays the execution time of random hot-deck imputation method on various categorical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	1
5	10
10	39

Table 5.15: Execution time for random hot-deck (categorical dataset)

5.2.3 k-NN (Mode)

Table 5.16 displays the execution time of k-NN (Mode) imputation method on various numerical datasets of different file size and for different values of k-Neighbors:

Dataset Size (MB)	k-Neighbors	Execution Time (sec)
0.5	3	34
	10	39
	20	49
1	3	106
	10	120
	20	128
2	3	142
	10	147
	20	158

Table 5.16: Execution time for k-NN (Mode)

5.2.4 Naive Bayes

Table 5.17 displays the execution time of Naive Bayes imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	2
5	7
10	14

Table 5.17: Execution time for Naive Bayes

5.2.5 Methods Comparison

Figure 5.2 displays the average execution time for all categorical imputation methods applied to a typical 1 MB dataset with 6.5% missing values.

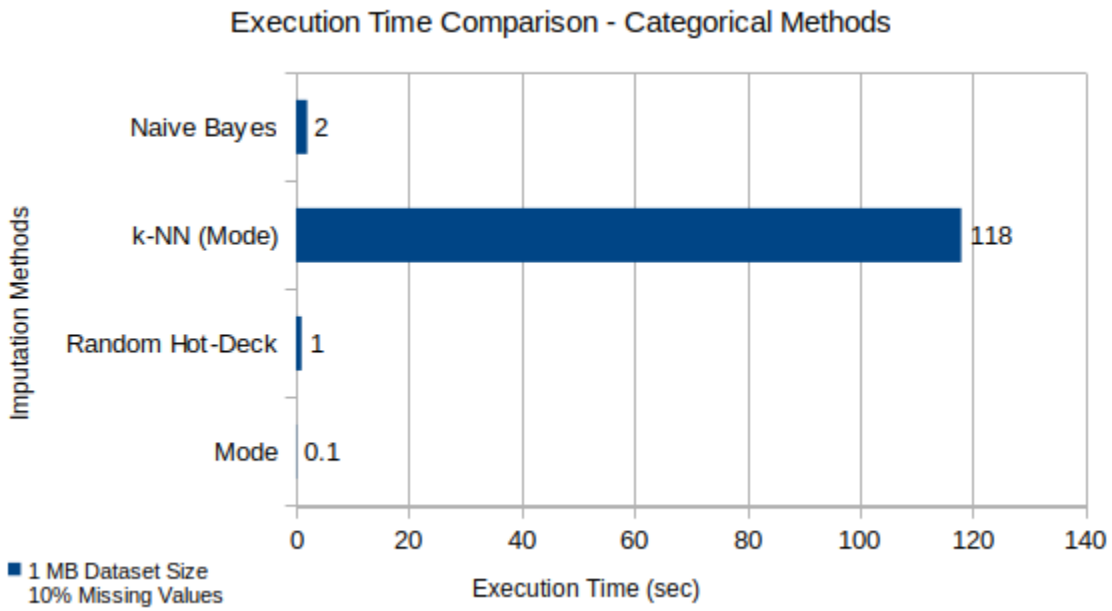

Figure 5.2: Execution Time Comparison - Categorical Methods

5.3 Mixed Datasets

5.3.1 Mean for Numerical - Mode for Categorical

Table 5.18 displays the execution time of Mean-Mode imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	0.2
5	0.9
10	1.7

Table 5.18: Execution time for Mean-Mode

5.3.2 Median for Numerical - Mode for Categorical

Table 5.19 displays the execution time of Median-Mode imputation method on various numerical datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	0.2
5	0.9
10	1.9

Table 5.19: Execution time for Median-Mode

5.3.3 Random Hot-Deck

Table 5.20 displays the execution time of random hot-deck imputation method on various mixed datasets of different file size:

Dataset Size (MB)	Execution Time (sec)
1	3
5	28
10	88

Table 5.20: Execution time for random hot-deck (mixed datasets)

5.3.4 Mixed k-NN (Mean for Numerical - Mode for Categorical)

Table 5.21 displays the execution time of mixed k-NN (Mean-Mode) imputation method on various numerical datasets of different file size and for different values of distance and k-Neighbors:

Dataset Size (MB)	Distance	k-Neighbors	Execution Time (sec)
0.2	Euclidean	3	31
		10	40
		20	53
	Manhattan	3	29
		10	37
		20	51
0.3	Euclidean	3	59
		10	75
		20	96
	Manhattan	3	59
		10	71
		20	95
1	Euclidean	3	633

Table 5.21: Execution time for mixed k-NN (Mean-Mode)

5.3.5 Mixed k-NN (Median for Numerical - Mode for Categorical)

Table 5.22 displays the execution time of mixed k-NN (Median-Mode) imputation method on various numerical datasets of different file size and for different values of distance and k-Neighbors:

Dataset Size (MB)	Distance	k-Neighbors	Execution Time (sec)
0.2	Euclidean	3	28
		10	43
		20	56
	Manhattan	3	28
		10	42
		20	60
0.3	Euclidean	3	68
		10	86
		20	98
	Manhattan	3	56
		10	74
		20	94
1	Euclidean	3	595

Table 5.22: Execution time for mixed k-NN (Median-Mode)

5.3.6 Methods Comparison

Figure 5.3 displays the average execution time for all mixed imputation methods applied to a typical 1 MB dataset with 6.5% missing values.

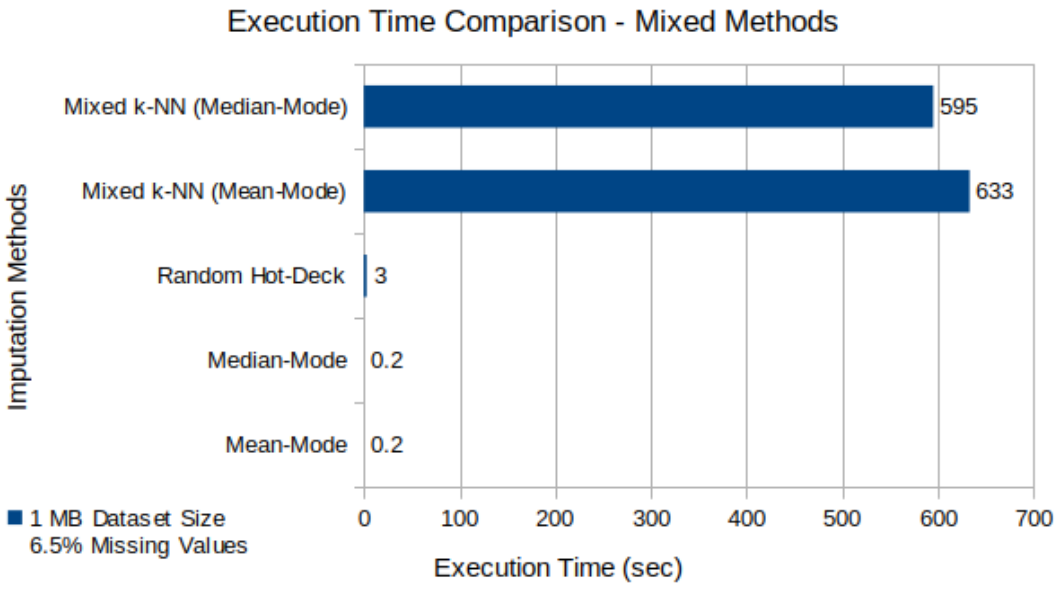

Figure 5.3: Execution Time Comparison - Mixed Methods

Chapter 6: Conclusion and Future Work

6.1 Conclusion

In this thesis, a web application for missing value imputation in datasets, named WEBIMPUTER, has been successfully developed. Datasets can be numerical, categorical or mixed. The application provides a user-friendly interface for users to upload their datasets with missing values, select appropriate imputation methods, and obtain the imputed datasets.

The available imputation methods for numerical datasets are: Mean, Median, Random Hot-Deck, Linear Regression, Lasso Regression, Ridge Regression and Support Vector Regression (SVR). For categorical datasets the available methods are: Mode, Random Hot-Deck, k-NN (Mode) and Naive Bayes. And for mixed datasets the available methods are: Mean-Mode, Median-Mode, Random Hot-Deck, Mixed k-NN (Mean-Mode) and Mixed k-NN (Median-Mode). The mathematical formulation and explanation of all these methods were also presented in this thesis.

The front-end of the application was written in HTML with use of CSS for styling the pages and JavaScript with jQuery for adding some functionalities on the client-side. Bootstrap was the framework utilized.

The main functionality of the application was implemented on the server-side. That includes all the modules that implement the imputation algorithms and all the processes that handle the receiving of the datasets with missing values and sending the imputed datasets back to client-side. The framework which was used for building the server-side is Flask and the programming language is Python.

Finally, a lot of experiments have been conducted by applying all the imputation methods of the application to various datasets of different file size and measuring the execution time, to help users gain a better understanding of the computational efficiency of the models. The results of the experiments have been presented both individually for each method and comparatively.

6.2 Future Work

A great number of features could be integrated into the application and a lot of different experiments could be conducted in the future.

For example some more advanced imputation methods that involve neural networks like Generative Adversarial Networks (GANs) [11] could be used. That would make the application more powerful and would probably attract more data scientists and engineers. However the server resources for such implementation should be taken into account, in order to avoid overloading.

As far as the experiments are concerned, it would be very useful to use various datasets with different missing value percentages and measure the accuracy and the performance of the imputation models [12]. In such case, a lot of insightful conclusions would be made regarding how missing values affect the effectiveness of the methods, so that users would have an indication of which imputation method should choose every time.

References

[1] I. B. Aydilek and A. Arslan, "A hybrid method for imputation of missing values using optimized fuzzy c-means with support vector regression and a genetic algorithm," *Information Sciences*, vol. 233, pp. 25–35, 2013.

[2] S. Reddy Sankepally, N. Kosaraju, and K. Mallikharjuna Rao, "Data imputation techniques: An empirical study using chronic kidney disease and life expectancy datasets," in *2022 International Conference on Innovative Trends in Information Technology (ICITIIT)*, pp. 1–7, 2022.

[3] D. W. Joenssen and U. Bankhofer, "Hot deck methods for imputing missing data," in *Machine Learning and Data Mining in Pattern Recognition* (P. Perner, ed.), (Berlin, Heidelberg), pp. 63–75, Springer Berlin Heidelberg, 2012.

[4] D. M. P. Murti, U. Pujianto, A. P. Wibawa, and M. I. Akbar, "K-nearest neighbor (k-nn) based missing data imputation," in *2019 5th International Conference on Science in Information Technology (ICSITech)*, pp. 83–88, 2019.

[5] A. Garcia and E. Hruschka, "Naive bayes as an imputation tool for classification problems," in *Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pp. 3 pp.–, 2005.

[6] T. Makaba and E. Dogo, "A comparison of strategies for missing values in data on machine learning classification algorithms," in *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, pp. 1–7, 2019.

[7] S. Shah, M. Telrandhe, P. Waghmode, and S. Ghane, "Imputing missing values for dataset of used cars," in *2022 2nd Asian Conference on Innovation in Technology (ASIANCON)*, pp. 1–5, 2022.

[8] M. Peña, P. Ortega, and M. Orellana, "A novel imputation method for missing values in air pollutant time series data," in *2019 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, pp. 1–6, 2019.

[9] L. Wijesekara and L. Liyanage, "Mind the large gap: Novel algorithm using seasonal decomposition and elastic net regression to impute large intervals of missing data in air quality data," *Atmosphere*, vol. 14, p. 355, 02 2023.

[10] V. Anandhi and R. Chezian, "Support vector regression to forecast the demand and supply of pulpwood," *International Journal of Future Computer and Communication*, vol. 2, pp. 266–269, 01 2013.

[11] S. C.-X. Li, B. Jiang, and B. Marlin, "Misgan: Learning from incomplete data with generative adversarial networks," 2019.

[12] W. Kim, W. Cho, J. Choi, J. Kim, C. Park, and J. Choo, "A comparison of the effects of data imputation methods on model performance," in *2019 21st International Conference on Advanced Communication Technology (ICACT)*, pp. 592–599, 2019.

[13] A. K.S., R. Ramanathan, and M. Jayakumar, "Impact of k-nn imputation technique on performance of deep learning based dfl algorithm," in *2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 153–157, 2021.

[14] L. Beretta and A. Santaniello, "Nearest neighbor imputation algorithms: a critical evaluation," *BMC Med. Inform. Decis. Mak.*, vol. 16 Suppl 3, p. 74, July 2016.

[15] K. Nishanth and R. Vadlamani, "Probabilistic neural network based categorical data imputation," *Neurocomputing*, vol. 218, 08 2016.

[16] J. Wang, D. Li, H. Zhang, X. Yu, A. Sekhari, Y. Ouzrout, and A. Bouras, "An improvement of support vector machine imputation algorithm based on multiple iteration and grid search strategies," in *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*, pp. 538–543, 2020.

[17] R. R. Andridge and R. J. A. Little, "A review of hot deck imputation for survey non-response," *International Statistical Review*, vol. 78, pp. 40–64, Apr. 2010.

[18] S. Thirukumaran and A. Sumathi, "Missing value imputation techniques depth survey and an imputation algorithm to improve the efficiency of imputation," in *2012 Fourth International Conference on Advanced Computing (ICoAC)*, pp. 1–5, 2012.

[19] T. Aljuaid and S. Sasi, "Proper imputation techniques for missing values in data sets," in *2016 International Conference on Data Science and Engineering (ICDSE)*, pp. 1–5, 2016.

[20] O. Harel and X.-H. Zhou, "Multiple imputation: review of theory, implementation and software," *Statistics in Medicine*, vol. 26, no. 16, pp. 3057–3077, 2007.