SUNY - STONY BROOK UNIVERSITY

Algorithms for Missing Data: MICE AND SICE

PROJECT REPORT - Group 7

Huynh Tan Vu	115317761
Brandon D'Anna	108470479

Stony Brook, NY, USA
December, 2022

Contents

Chapter 1

Motivation and Background

In a society with an ever increasing global datasphere, higher volumes of data increase the probability of data integrity vulnerabilities. Many of the most popular machine learning techniques ranging from neural networks to support vector machines, cannot be applied on datasets with missing values. Additionally, missing data can lead to performance degradation while incorrect imputations can result in subpar models and subsequent predictions (Khan, 2020 [1]).

Alternative methods for data imputation exist, many with drawbacks that render them less optimal than those presented. Deletion is both one of the most naive and least efficient methods, resulting in significant loss of information and decreased statistical power. A popular method utilizes summary statistics, most commonly the mean, median, or mode to impute missing values. While more efficient than deletion, this method fails to consider the inherent uncertainty of imputed values. Recognizing imputed values as real values can cause bias in the resulting model and its corresponding outputs. Lastly, K-Nearest Neighbors (KNN) can have degraded accuracy with high-dimensional data.

Multivariate Imputation by Chained Equation (MICE) algorithm is popular for missing data imputation. In this algorithm, each variable's corresponding missing values are imputed using the mean. Next, each variable's missing values are reverted back to missing and a linear regression model is built for the given variable. The resulting model is then used to predict the variable's missing values. This process is repeated until all n variables have had a linear regression model built using the n-1 remaining variables. Once completed, the table is stored as an iteration. The process is then repeated again, in various random orders until all of the resulting iteration tables converge to the same predicted values for each respective variable's missing observations.

Chapter 2

Related Work

Khan S. and Hoque [1], A. proposed an extension of MICE, computing the performance of the proposed algorithm with the existing MICE and found the former achieved high accuracy, F-measure, and less error than its competitors (Khan, 2020 [1]). The proposed algorithm, Single Center Imputation from Multiple Chained Equations (SICE) uses MICE m times, with m representing the user-defined number of iterations. The results are stored in an array. Each missing value is then replaced with the mean for numeric variables and mode for binary variables. This research is most closely related to the work presented in this project.

The aforementioned algorithm was performed in the R environment using the algorithms of the MICE package. This package implements MICE with the ability to choose from various algorithms including but not limited to Lasso, Logistic, Linear, and Quadratic Regression. This library also includes a function, ampute, which generates multivariate missing data (Van Buuren, 2011 [2]). Similarly, there exists a method in the Scikit-learn library in the Python programming language for implementing multivariate feature imputation. Neither of these libraries have implemented the SICE algorithm.

Chapter 3

Contribution

A python library was created from scratch to incorporate both MICE and SICE using a function with various parameters. While the MICE library in the R environment includes several regression based supervised machine learning algorithms, we expanded the algorithm options to include Random Forest. This algorithm is widely used in both classification and regression variable predictions. Random Forest is built upon the creation of multiple decision trees. Each of these trees computes a prediction for the given observation and the majority vote is taken in the case of binary variables and the mean is taken in the case of continuous variables.

As mentioned in the Related Works, neither library in the R or Python environment have implemented the SICE algorithm. Our library incorporates a parameter that takes the value "MICE " or "SICE". This, in combination with the number of iterations parameter for the latter choice, allows the user to utilize the SICE algorithm with a certain number of iterations. As will be discussed in greater detail in the Results section, SICE is more efficient than MICE both in terms of computational cost and prediction accuracy.

Chapter 4

Details about how contributions were made

To begin, the contribution for the whole project mostly focuses on 2 main parts, with the first part focusing on building MICE and SICE from scratch, while the second part improves and extends the original algorithms, namely Linear Regression, with other Machine Learning algorithms for binary and continuous data.

Firstly, even though there are some MICE libraries that exist and are more optimized than the library that we are building, a SICE library has not yet been built and widely distributed. Therefore, our team decided to first build the MICE package from scratch and later extend to SICE. Moreover, the built-from-scratch library would be more user-friendly, and easier for specifying the parameter inputs. For example, the SICE algorithms would allow users to pre-specify not only the number of iterations but also the choice of underlying algorithms for binary or continuous data, such as Random Forest or Lasso Regression. Those Python libraries would be constructed in an accessible manner, which facilitates the process of modifying, extending and upgrading for later use.

Secondly, most of the existing library was constructed around the Linear Regression algorithm. However, the fact that continuous and binary data have different properties can be a serious problem when it comes to applying just Linear regression, which can mislead the predictions and thus the results of data imputation. Therefore, in this project, the index column of binary data would be saved and treated with other specific classification algorithms. As a result, the results for the imputation would be expected to be more accurate and reasonable. Moreover, the library is extended with other Machine Learning algorithms. In specific, while the continuous data would be fitted with Linear regression and Lasso regression, Logistics regression and Random Forest would be applied for the binary data.

Chapter 5

Empirical Results

5.1 Performance Measures

Different functions are used in this empirical study in terms of time and the accuracy of the imputation. The root mean squared error (RMSE) are applied to measure the precision of imputation forecasting. This provides an indication on how far the predicted values are from the actual values. Hence, using this measure would give a better vision of how good the forecast is. The measurement metric is given as follows:

$$\text{RMSE} = \sqrt{\frac{\sum_{t=1}^{T}(\hat{y}_t - y_t)^2}{T}}$$

where T is number of missing data

Lastly, the empirical imputation time of each measurement is also recorded for later time efficiency comparison.

5.2 Empirical Results

First, to facilitate the process of testing the efficiency of those algorithms, sample data will contain 15 columns and 5000 rows. Moreover, from this full data, the sample missing data with 10% missing rate would be constructed for MICE and SICE tests. In this experiment, while the default number of iteration for SICE will be set to 5, the convergence error threshold for MICE is 0.001

The table below summarizes all the performance measures across the various methods:

Method	Algorithm	Iterations	Sum of Squared Error
Linear Regression/Random Forest	SICE	5	6079.73
Linear Regression/Random Forest	MICE	14	6136.54
Lasso and Logistic Regression	SICE	5	6081.96
Lasso and Logistic Regression	MICE	3	6081.96
Linear and Logistic Regression	SICE	5	6115.18
Linear and Logistic Regression	MICE	6	6122.32
Lasso Regression/Random Forest	SICE	5	6034.43
Lasso Regression/Random Forest	MICE	10	6132.83

There are some notable details that partly show the efficiency of MICE in terms of time and accuracy. The SSE of SICE was slightly better than that of MICE algorithm even though the number of iterations of MICE is higher than than of SICE. In specific, For the LR/RF combination, the SSE of MICE was nearly 100 higher than that of SICE, while it takes 15 iterations for MICE to converge, which was nearly 3 times higher than SICE's iteration (5). The same pattern also happened with other combinations such as LassoR/RF and LR/LogR.

Method	Algorithm	Iterations	Computation Time (Seconds)
Linear Regression/Random Forest	SICE	5	46.6
Linear Regression/Random Forest	MICE	14	130.6
Lasso and Logistic Regression	SICE	5	1.25
Lasso and Logistic Regression	MICE	3	0.75
Linear and Logistic Regression	SICE	5	1.19
Linear and Logistic Regression	MICE	6	1.38
Lasso Regression/Random Forest	SICE	5	49.22
Lasso Regression/Random Forest	MICE	10	93.10

Moreover, the computational time of SICE algorithms is superior to that of MICE. This can be also partly explained with the number of iterations required for MICE's errors to converge. To be clear, the computational time for SICE was approximately 2-3 times faster than that of MICE, 46.6 seconds and 130 seconds for the LR/RF combination and 49.22 seconds and 93.10 seconds for the LassoR/RF combination, respectively. As a result, for most combinations, the SSE and computational time of SICE was slightly to surprisingly better than those of MICE, which empirically concludes the superior of the new former to the latter.

Chapter 6

Limitations and Future Direction

6.1 Limitation

There are several limitations that should be considered in this project. First, the MICE and SICE libraries are built from scratch, which mostly utilize the hashmap data structure to improve the time complexity (maximum of $O(ncol * nrow)$). However, in return, the space complexity could be a tremendous problem since all the iteration data is stored in dictionaries and further extended to 4 dimensional hashmap. Therefore, later attention should be paid on this trade-off or actual algorithms to reduce both time and space complexity. Second, although other state-of-the-art algorithms could be integrated into this library, the stability of those algorithms still remains mysterious. In specific, Deep Learning algorithms could be extremely good for binary classification, but the cost function is not convex, which results in different parameters and models for each iteration and thus affects the quality of data prediction. Therefore, this instability partly explained why we mostly adopted originally efficient Machine Learning models rather than resorting to highly complicated ones.

6.2 Future Direction

As proposed above, the improvement of space complexity would be tremendously beneficial for the MICE and SICE algorithm when it comes to big data of million rows and columns. Moreover, the above algorithms could be further improved with other efficient ML algorithms, such as SVMs or other statistical regression methods. If the stability of some state of the art models could be monitored and controlled, it is also highly recommended for the library extension but the potential problems of overfitting should be avoided. However, beside accuracy, the underlying algorithms should rather concentrate on computational efficiency and interpretability. While the former was mentioned and suggested for improvement above, the latter could be enhanced with some feature engineering methods. In such, the potentially actual features for data prediction could be filtered,

thus having more predictive power and better accuracy. Therefore, the deployment of some feature engineering techniques could be a potential direction to improve MICE and SICE algorithms.

Chapter 7

Workload Division

Both team members contributed equally to the initial research and ultimate decision to pursue the topic at hand. The slides for the initial presentation, midpoint update, and final update were split evenly between the two. The initial code structure for the two algorithms was constructed by Vu. Brandon added in code for alternative algorithms for both binary and continuous variables and parameterized the algorithm to respond to user input. Code to detect whether a given variable is continuous or binary was also added. Vu debugged the final code. Brandon constructed a sample dataset. This dataset was created with two versions, one with all values present and one with 10% of cells randomly assigned to be missing. Both team members tested the function with various inputs and created a comparison table of results.

Bibliography

[1] Khan, S. I., Hoque, A. S. M. L. (2020). SICE: an improved missing data imputation technique. Journal of big data, 7(1), 37. https://doi.org/10.1186/s40537-020-00313-w

[2] Van Buuren S, Groothuis-Oudshoorn K (2011). "mice: Multivariate Imputation by Chained Equations in R." Journal of Statistical Software, 45(3), 1-67. doi:10.18637/jss.v045.i03.

Appendix A

Mice Code

Listing A.1: Python based MICE library

```
import numpy as np
from copy import deepcopy
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression, Lasso
from sklearn.ensemble import RandomForestClassifier
import statistics as st
import time



class MICE:
    def __init__(self, data, binary_set, bin_algorithm, cont_algorithm, threshold):
        self.X = data
        self.binary_set = binary_set
        self.bin_algorithm = bin_algorithm
        self.cont_algorithm = cont_algorithm
        self.threshold = threshold

        self.nan_dic = {i: [] for i in range(self.X.shape[1])}
        self.nan_dic = self.nan_missing_create(self.nan_dic, type='nan_dic')

        self.fitted_X = deepcopy(self.X)


    def nan_missing_create(self, target_dic, type='nan_dic'):
        for i in range(self.X.shape[1]):
            for j in range(self.X.shape[0]):
                if np.isnan(self.X[j, i]):
                    if type == 'nan_dic':
                        target_dic[i].append(j)
```

```
                else:
                    # 2d list
                    # 2nd element: prediction, 3rd element: error
                    target_dic[i].append([j, 0, 0])
        return target_dic

        # self.nan_dic[i].append(j)

    # use for first iteration
    def place_holder(self, X_sample, option='MEAN'):
        n_col = X_sample.shape[1]
        for col in range(n_col):
            if option == 'MEAN' and col not in self.binary_set:
                mean = np.nanmean(X_sample[:, col])
                for i in self.nan_dic[col]:
                    X_sample[i, col] = mean

            elif option == 'MEDIAN' and col not in self.binary_set:
                med = np.nanmedian(X_sample[:, col])
                for i in self.nan_dic[col]:
                    X_sample[i, col] = med

            else:
                mode = st.mode(X_sample[:, col])
                for i in self.nan_dic[col]:
                    X_sample[i, col] = mode

        return X_sample

    def LR_one_iter_fit(self, current_dataset, it_dic):
        # it_dic: # 2d dic second element in this dic[list] to store value of prediction
        max_error = 0
        # make sure there is no nan in this current_dataset
        # copy dataset: taking out value for further prediction
        copy_dataset = deepcopy(current_dataset)

        # final_ori_dataset: replace predicted value to this dataset
        final_ori_dataset = deepcopy(current_dataset)
        n_col = copy_dataset.shape[1]

        for col in range(n_col):

            # temp remove all missing data of col_i
            sub_dataset = np.delete(current_dataset, self.nan_dic[col], 0)
            y = sub_dataset[:, col]
            X_ = np.delete(sub_dataset, col, 1)
```

```
            if col in self.binary_set:
                if self.bin_algorithm == 'random_forest':
                    # creating a RF classifier
                    reg = RandomForestClassifier(n_estimators=100)
                    reg.fit(X_, y)
                else:
                    reg = LogisticRegression().fit(X_, y)
            else:
                if self.cont_algorithm == 'linear_regression':
                    reg = LinearRegression().fit(X_, y)
                else:
                    reg = Lasso().fit(X_, y)

            # reg = LinearRegression().fit(X_, y)

            # take original X without col_i
            sub_X = np.delete(copy_dataset, col, 1)

            for i in range(len(self.nan_dic[col])):
                missing_row = self.nan_dic[col][i]
                prediction_value = reg.predict(sub_X[missing_row, :].reshape(-1, 1).T)

                # replace to final_ori_data not copy
                final_ori_dataset[missing_row, col] = prediction_value

                pre_prediction_value = it_dic[col][i][1]
                if col not in self.binary_set:
                    max_error = max(max_error, abs(prediction_value - pre_prediction_value))

                # save this prediction, error value to this iteration_dic:
                it_dic[col][i][1] = prediction_value
                it_dic[col][i][2] = abs(prediction_value - pre_prediction_value)

        return final_ori_dataset, it_dic, max_error

    def MICE_fit(self):
        max_error = 10e9
        # just set for the initial time
        self.fitted_X = self.place_holder(X_sample=self.fitted_X)
        i = 0
        MICE_iter_dic = {j: [] for j in range(self.X.shape[1])}
        # 2nd element: prediction, 3rd element: error
        MICE_iter_dic = self.nan_missing_create(MICE_iter_dic, type='other')
        dif_error = 10e9
        while dif_error > self.threshold:
            print(f'iteration {i + 1}')
            self.fitted_X, MICE_iter_dic, max_it_error = self.LR_one_iter_fit(self.fitted_X,
```

```
            MICE_iter_dic)
            print(max_it_error)
            dif_error = abs(max_it_error - max_error)
            print(dif_error)
            max_error = max_it_error
            i += 1

def make_binary_set(data):
    bin_set = set()
    new_data = data.dropna()
    for col in range(new_data.shape[1]):
        if np.array_equal(new_data.iloc[:, col], new_data.iloc[:, col].astype(bool)):
            bin_set.add(col)
    return bin_set

if __name__ == '__main__':
    path = '/Users/tanvu10/Downloads/'
    missing_data_file_name = 'sample (1).csv'
    full_data_file_name = 'sample_full (1).csv'

    full_data_set = pd.read_csv(path + full_data_file_name)
    full_data_set = full_data_set.iloc[:, 1:]
    full_data_set = full_data_set.to_numpy()

    missing_data_set = pd.read_csv(path + missing_data_file_name)
    missing_data_set = missing_data_set.iloc[:, 1:]
    bin_set = make_binary_set(missing_data_set)
    missing_data_set = missing_data_set.to_numpy()
    # print(missing_data_set)
    mice_model = MICE(data=missing_data_set, binary_set=bin_set
                      , bin_algorithm='random_forest', cont_algorithm='linear_regression',
                      threshold=10**(-4))
    # sice_model = SICE(data=missing_data_set, n_iteration=10, binary_set=bin_set
    #                   , bin_algorithm='log_regression', cont_algorithm='lasso_regression')
    t0 = time.time()
    mice_model.MICE_fit()
    t1 = time.time() - t0
    print("Time elapsed: ", t1)
    # sice_model.SICE_result()
    print(pd.DataFrame(mice_model.fitted_X))
    error = 0
    n_col = full_data_set.shape[1]
    for col in range(n_col):
        for row in mice_model.nan_dic[col]:
            error += (mice_model.fitted_X[row, col] - full_data_set[row, col]) ** 2
    print(error)
```

Appendix B

Sice Code

Listing B.1: Python based SICE library

```
import numpy as np
from copy import deepcopy
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression, Lasso
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LassoCV
from sklearn.model_selection import RepeatedKFold
import statistics as st
import time
class SICE:
    def __init__(self, data, n_iteration, binary_set, bin_algorithm, cont_algorithm):
        self.X = data
        self.iteration = n_iteration
        self.nan_dic = {i: [] for i in range(self.X.shape[1])}
        self.nan_dic = self.nan_missing_create(self.nan_dic, type='nan_dic')

        self.result_dic = {} # (row, col) = []
        self.result_dic = self.nan_missing_create(self.result_dic, type='result')

        self.fitted_X = deepcopy(self.X)
        self.SICE_value_dic = {}

        self.binary_set = binary_set
        self.bin_algorithm = bin_algorithm
        self.cont_algorithm = cont_algorithm

    def nan_missing_create(self, target_dic, type='nan_dic'):
        for i in range(self.X.shape[1]):
            for j in range(self.X.shape[0]):
                if np.isnan(self.X[j, i]):
```

```
                if type == 'nan_dic':
                    target_dic[i].append(j)
                elif type == 'result':
                    target_dic[(j, i)] = [] # row, col
                else:
                    # 2d list
                    # second element in this list to store error or value of prediction
                    target_dic[i].append([j, 0])
        return target_dic

                # self.nan_dic[i].append(j)

    # use for first iteration
    def place_holder(self, X_sample, option='MEAN'):
        n_col = X_sample.shape[1]
        for col in range(n_col):
            if option == 'MEAN' and col not in self.binary_set:
                mean = np.nanmean(X_sample[:,col])
                for i in self.nan_dic[col]:
                    X_sample[i, col] = mean

            elif option == 'MEDIAN' and col not in self.binary_set:
                med = np.nanmedian(X_sample[:, col])
                for i in self.nan_dic[col]:
                    X_sample[i, col] = med

            else: # auto MODE for binary data
                mode = st.mode(X_sample[:, col])
                for i in self.nan_dic[col]:
                    X_sample[i, col] = mode

        return X_sample

    def LR_one_iter_fit(self, current_dataset, it_dic):
        # it_dic: # 2d dic second element in this dic[list] to store value of prediction

        # make sure there is no nan in this current_dataset
        # copy dataset: taking out value for further prediction
        copy_dataset = deepcopy(current_dataset)

        # final_ori_dataset: replace predicted value to this dataset
        final_ori_dataset = deepcopy(current_dataset)
        n_col = copy_dataset.shape[1]

        for col in range(n_col):

            # temp remove all missing data of col_i
```

```
            sub_dataset = np.delete(current_dataset, self.nan_dic[col], 0)
            y = sub_dataset[:, col]
            X_ = np.delete(sub_dataset, col, 1)

            if col in self.binary_set:
                if self.bin_algorithm == 'random_forest':
                    # creating a RF classifier
                    reg = RandomForestClassifier(n_estimators=100)
                    reg.fit(X_, y)
                else:
                    reg = LogisticRegression().fit(X_, y)
            else:
                if self.cont_algorithm == 'linear_regression':
                    reg = LinearRegression().fit(X_, y)
                else:
                    reg = Lasso().fit(X_, y)

            # take original X without col_i
            sub_X = np.delete(copy_dataset, col, 1)

            for i in range(len(self.nan_dic[col])):
                missing_row = self.nan_dic[col][i]
                prediction_value = reg.predict(sub_X[missing_row, :].reshape(-1, 1).T)

                # print(missing_row, col)
                # replace to final_ori_data not copy
                final_ori_dataset[missing_row, col] = prediction_value

                # save this prediction value to this iteration_dic:
                it_dic[col][i][1] = prediction_value

        return final_ori_dataset, it_dic

    def SICE_fit(self):
        # just set for the initial time
        self.fitted_X = self.place_holder(X_sample=self.fitted_X)
        sample_MICE_iter_dic = {j: [] for j in range(self.X.shape[1])}
        sample_MICE_iter_dic = self.nan_missing_create(sample_MICE_iter_dic, type='other')

        for i in range(self.iteration):
            print(f'iteration_{i+1}')
            MICE_iter_dic = deepcopy(sample_MICE_iter_dic)
            self.fitted_X, MICE_iter_dic = self.LR_one_iter_fit(self.fitted_X, MICE_iter_dic)
            self.SICE_value_dic[i] = MICE_iter_dic

    def SICE_result(self):
        n_col = self.fitted_X.shape[1]
```

```
        for i in range(self.iteration): # self.iteration is small (guarantee in O(n^2))
            iter_dic = self.SICE_value_dic[i]

            for col in range(n_col):
                for j in range(len(self.nan_dic[col])):
                    row = iter_dic[col][j][0]
                    iter_result = iter_dic[col][j][1]
                    self.result_dic[(row, col)].append(iter_result)

        for row, col in self.result_dic.keys():
            self.fitted_X[row, col] = np.mean(self.result_dic[(row, col)])


def make_binary_set(data):
    bin_set = set()
    new_data = data.dropna()
    for col in range(new_data.shape[1]):
        if np.array_equal(new_data.iloc[:, col], new_data.iloc[:, col].astype(bool)):
            bin_set.add(col)
    return bin_set

if __name__ == '__main__':
    path = '/Users/tanvu10/Downloads/'
    missing_data_file_name = 'sample (1).csv'
    full_data_file_name = 'sample_full (1).csv'

    full_data_set = pd.read_csv(path + full_data_file_name)
    full_data_set = full_data_set.iloc[:, 1:]
    full_data_set = full_data_set.to_numpy()

    missing_data_set = pd.read_csv(path + missing_data_file_name)
    missing_data_set = missing_data_set.iloc[:, 1:]
    bin_set = make_binary_set(missing_data_set)
    missing_data_set = missing_data_set.to_numpy()
    # print(missing_data_set)
    sice_model = SICE(data=missing_data_set, n_iteration=5, binary_set=bin_set
                      ,bin_algorithm='random_forest', cont_algorithm='linear_regression')
    # sice_model = SICE(data=missing_data_set, n_iteration=10, binary_set=bin_set
    #                   , bin_algorithm='log_regression', cont_algorithm='lasso_regression')
    t0 = time.time()
    sice_model.SICE_fit()
    t1 = time.time() - t0
    print("Time elapsed: ", t1)
    sice_model.SICE_result()
    print(pd.DataFrame(sice_model.fitted_X))
    error = 0
    n_col = full_data_set.shape[1]
```

```
for col in range(n_col):
    for row in sice_model.nan_dic[col]:
        error += (sice_model.fitted_X[row, col] - full_data_set[row,col])**2
print(error)
```