

Missing Data Imputation Using Gray KNN
-By Nikhil Sukhdev

Table of Contents

1. Introduction.....	3
1.1 General Introduction	3
1.2 Advantages of GkNN	3
2. Overview of The Project.....	4
2.1 The GkNN algorithm.	4
2.2 Gray Distances for kNN.....	4
2.3 Pseudo Code and Its Implementation.....	5
3. Complexity Analysis	7
4. Implementation	9
5. Experimental Results.....	13
6. References.....	17

1. Introduction

1.1 General Introduction

The GkNN algorithm is a variation of the kNN algorithm and makes use of gray distances to impute missing values in instances. The algorithm has certain advantages that makes it much superior to the regular kNN algorithm one of which is that it can impute missing values for heterogeneous data really well. Therefore, it is now extensively being used in many government labs that were currently making use of the kNN algorithm [1].

1.2 Advantages of GkNN

The GkNN algorithm was developed with the view of overcoming the problems that the kNN algorithm faced. The algorithm was for the most part successful but it had the inadvertent effect of increasing the time complexity of the kNN algorithm and hence can only be used in certain conditions. The advantages of GkNN over the aforementioned imputation techniques are as follows.

1. One of the major drawbacks of single imputation techniques was the fact that it was not able to provide valid confidence intervals as well as standard errors. GkNN can not only overcome this major flaw by providing the above-mentioned parameters but can also contain factors such as implicit uncertainty.
2. The GkNN algorithm also makes use of all the observed instances, which means it will take into consideration all the values including those instances that are missing. The two previously stated methods fall short here as they only take into consideration all the complete instance while disregarding the incomplete ones.
3. One of the main drawbacks of the kNN algorithm was its inability to handle data that was heterogeneous in nature that is contained both numeric as well as categorical type data. GkNN algorithm has successfully overcome this issue, hence making it different from C4.5 algorithm as well as kernel methods.

2. Overview of The Project

The project makes use of kNN for the most part, the only difference being that it used gray distances (derived from GRA that depends on GRC and GRG).

2.1 The GkNN algorithm

The NN algorithm, the fore runner to the now widely popular kNN algorithm was a technique that was widely used for the data for which no prior information was available. The NN algorithm was prone to many problems such as overfitting, a problem that was carried forward to its successor kNN [2]. The kNN algorithm makes use of a simple technique known as mean/mode imputation technique, wherein if the data is categorical, the missing value is replaced by mode and mean if the data is numerical. Mathematically, this can be represented as

$$Y = \begin{cases} \arg \max_v \left\{ \sum_{(X_j, Y_j, 1) \in D_k} 1(Y_j = v) \right\}, & \text{if } Y \text{ is categorical} \\ \frac{1}{k} \sum_{j=1}^k Y_j, & \text{if } Y \text{ is numerical} \end{cases}$$

The kNN algorithm is an instance based lazy learner and relies only on the values imputed in it. The main challenge in kNN is to find the value of k. If k is too small, it may not be able to see the majority of values surrounding it and if it is too large, it will simply consider the mode of the entire dataset. A novel approach to overcome this issue was proposed [3], wherein the value of $k = \sqrt{n}$ was suggested where n is the total number of instances, however this was only valid for instances > 100

2.2 Gray Distances for kNN

Gray Relational analysis was devised in 1989 by Deng [4]. In engineering applications, the jargon Black Box means the data or information about a certain system or function is unavailable and in contrast to that, White Box means a system for which all the values and information is known.

The Gray KNN imputation algorithm works on the concept of gray distances. The gray distances are quantified using the gray relational analysis [5-6], which is used to measure the relationship between two instances of a dataset. The gray relational coefficient (GRC) is used to estimate the closeness of the instances. A dataset with n instances and m attributes is taken. The complete instances are called as observed instances and the instances with missing values are called as reference instances. The GRC is defined below:

$$GRC(x_0(p), x_i(p)) \frac{dy}{dx} = \frac{\min_j \min_k |x_0(k) - x_j(k)| + \rho \max_j \max_k |x_0(k) - x_j(k)|}{|x_0(p) - x_i(p)| + \rho \max_j \max_k |x_0(k) - x_j(k)|} \quad (1)$$

$i=j=1, 2, \dots, n$ is the number of instances and $k=p=1, 2, \dots, m$ is the number of attributes. x_0 is the reference instance with missing data, x_i is the observed instance. $\rho \in [0, 1]$ is called the distinguishing coefficient. In this research paper, ρ is taken as 1.

Hence, using the GRC formula above, the similarity between the attributes of the observed and the reference instance is compared. As a rule of thumb, we consider the GRC between 2 instances for a given attribute as 0 if they are dissimilar with each other. On the other hand, we consider the GRC as 1, if they are completely like each other. As it is impractical to compare the similarity of every attribute with their respective observed instances, a GRG (Gray Relational Grade) has been formulated. The GRG takes the mean of all the GRC's for every instance. The GRG is given as follows:

$$GRG(x_0, x_i) = \frac{1}{m} \sum_{k=1}^m GRC(x_0(k), x_i(k)), i = 1, 2, \dots, n \quad (2)$$

Once the GRG between the reference instance and each observed instance is found, we sort them in descending order. The pair with the highest GRG is most similar and likewise, the pair with lowest GRG is most dissimilar. Once the sorting is done, depending on the value of k, the instances with highest GRG (most similar) are chosen for the imputation.

2.3 Pseudo Code and Its Implementation

The algorithm, especially in a pseudo-code format helps understanding the algorithm simply. The algorithm is broken down line by line and syntax by syntax to make the process much simpler.

Part – 1

First Iteration

FOR each instance in the dataset:

Classify based on class label

The first line of code calls for all the instances in the dataset, and then classifies them based on class label, discrete values whose values are directly or indirectly dependent on the values of other attributes. It is important to note that the algorithm calls for instances of a dataset rather than features, i.e., columns.

In plain language, it can be said that the first two lines initiate the dataset.

Part – 2

FOR each class:

FOR each attribute j in X_c :

Switch (type of j)

Case Discrete Mode: {attribute j in c}

Case Symbol Mode: {attribute j in c}

Case Continuous Mean: {attribute j in c}

The code at first initiates all classes available in the dataset after which it goes on to initiate each attribute in said classes. Then the SWITCH statement is initiated, which is like an if – else statement of sorts. This statement has three cases.

Case Discrete Mode: {attribute j in c}

Case Discrete Model as the name suggests at first considers the mode in each given class of discrete values. This is the first case as while doing basic KNN analysis, the attribute with complete repeating values is matched with those that have missing values. An important factor to consider is this is only valid for categorical type data as for numeric type data, mean is taken into consideration.

Case Symbol Mode: {attribute j in c}

This case considers only symbolic data and then calculates its mode as the mean of such data is not possible. An example of symbolic data can be DNA sequences or a grade given to certain fruits and vegetables.

Case Continuous Mean: {attribute j in c}

Case Continuous mean considers the mean of the numeric data. The mode, as mentioned earlier cannot be considered as it gives inaccurate results.

Part – 3

FOR each MV (i, j) in Ti:

SWITCH (type of j)

CASE Discrete: $MV_{(i,j)} = \text{Mode } \{\text{attribute j in c}\}$

CASE Symbol: $MV_{(i,j)} = \text{Mode } \{\text{attribute j in c}\}$

CASE Continuous: $MV_{(i,j)} = \text{Mean } \{\text{attribute j in c}\}$

This follows the same trope as mentioned above, the only difference being it is considering only the missing values (denoted by $MV_{(i,j)}$) rather than taking all attributes and then equating those values with mean, median and mode respectively.

// tth iteration

REPEAT

// Grey Relational Analysis

FOR each class D_i :

FOR each instance in X_1 :

 Compute $GRG_{(i,j)}$ (based on formula (2))

 Sorting $GRG_{(i,j)}$

 //Get k Nearest Instances

The above-mentioned algorithm is a basic representation of the Grey KNN algorithm that is shown above.

Find k maximal $GRG_{(i,j)}$

Deriving k instances associated with k maximal $GRG_{(i,j)}$

FOR each instance obtained from Step 2.3.2

 get the value of attribute j: $X_c = C_t(1, j) \dots C_t(i, j)$

This step does the job of finding k maximal GRG by deriving all the instances that are associated with the k maximal of the same.

//Imputation $CIMV_{t(i,j)}$

3.4.1 SWITCH (type of j)

CASE Discrete: $MV_{(i,j)} = \text{Mode} \{ \text{attribute } j \text{ in } X_e \};$

CASE Symbol: $MV_{(i,j)} = \text{Mode} \{ \text{attribute } j \text{ in } X_c \};$

CASE Continuous: $MV_{(i,j)} = \text{Mean} \{ \text{attribute } j \text{ in } X_c \};$

UNTIL (convergence or $t \leq 0$)

3. Complexity Analysis

Let m be the number of instances compared during the imputation process and n be the number of attributes in each dataset. First step in GKNN imputation is to replace all NaN or missing values with either mean or mode depending on the type of attribute. Below is the pseudo code snippet for the same.

```

2.2.1 FOR each class
2.2.2   FOR each attribute j in  $X_c$ 
2.2.3     SWITCH (type of j)
           CASE Discrete:  $\text{Mode}\{\text{attribute } j \text{ in } X_c\}$ 
           CASE Symbol:  $\text{Mode}\{\text{attribute } j \text{ in } X_c\}$ 
           CASE Continuous:  $\text{Mean}\{\text{attribute } j \text{ in } X_c\}$ 
2.3.1   FOR each  $MV(i,j)$  in  $T_i$ 
2.3.2     SWITCH (type of j)
           CASE Discrete:  $\hat{MV}_1(i,j) = \text{Mode}\{\text{attribute } j \text{ in } X_c\};$ 
           CASE Symbol:  $\hat{MV}_1(i,j) = \text{Mode}\{\text{attribute } j \text{ in } X_c\};$ 
           CASE Continuous:  $\hat{MV}_1(i,j) = \text{Mean}\{\text{attribute } j \text{ in } X_c\};$ 

```

The time complexity of mean/mode imputation is $O(m * n)$. The GKNN algorithm uses GRG (Gray Relational Grade) to estimate the similarities and dissimilarities between observed and reference instance. The pair with highest GRG is most similar while the pair with lowest GRG is most dissimilar. To calculate GRG, algorithm iterates through all the classes (D) and creates an inner for loop to iterate through each instance (X) which takes time complexity $O(m * n)$ which means the time complexity increases linearly and in direct proportion to number of inputs m & n in this case [7].

```

3.2.1 FOR each class  $\hat{D}_i$ 
3.2.2   FOR each instance in  $X_i$ 
3.2.3     Compute  $\text{GRG}(i,j)$  base on formula (2)

```

Once GRG is obtained further computations are done to sort GRG in descending order to find most similar pairs (reference & observed). The time complexity for sorting is generally higher than $O(m * \log(m))$, Usually $O(m * \log(m))$ is observed in algorithms where data is repeatedly divided into half and each half is processed again independently. For example, merge sort, heapsort [8].

```

3.2.4   Sorting  $\text{GRG}(i,j)$ 

```

After sorting GRG, algorithm finds k neighbours or pairs with most similarities, in our experiments we found $k = 9$ gives the most optimum results (low NRMS).

```

3.3           //Get k Nearest Instances
3.3.1 Find k maximal GRG( $i, j$ )
3.3.2 Deriving  $k$  instances associated with k maximal GRG( $i, j$ )

```

Adding all these time complexities of mean/mode imputation, GRG calculation, GRG sorting the overall time complexity of imputing whole dataset without classification comes out to be $O(k_2 * m^2 * n * \log(m))$ where k_2 is the number of imputation iterations. It is a very small value usually. $K_2 = 2$ & $K_2 = 1$ in our experiments. For best results (low run time & low NRMS) k_2 should be 2.

If the dataset has k_1 classes, as our algorithm performs the KNN method independently on each cluster for missing value imputation, the complexity of the GKNN algorithm is $O(k_1 * k_2 * n * m_j^2 \log(m_j))$, where m_j is the number of instances in the biggest class j , i.e., class j is the class containing the maximal number of instances.

Generally speaking, m_j is lesser than n when $k_1 > 1$, so we have $O(k_1 * k_2 * n * m_j^2 * \log(m_j)) < O(k_2 * n * m^2 * \log(m))$ [9]. The computation cost in the GKNN algorithm is less than traditional imputation algorithms, such as the KNN algorithm.

```

3.0 // tth iteration
3.1 REPEAT
3.2     // Grey Relational Analysis
3.2.1 FOR each class  $D_i$ 
3.2.2     FOR each instance in  $X_i$ 
3.2.3         Compute GRG( $i, j$ ) base on formula (2)
3.2.4     Sorting GRG( $i, j$ )
3.3     //Get k Nearest Instances
3.3.1 Find k maximal GRG( $i, j$ )
3.3.2 Deriving  $k$  instances associated with k maximal GRG( $i, j$ )
3.3.3 FOR each instance obtained from Step 2.3.2
3.3.4     get the value of attribute  $j$ :  $X_C = C_i^*(1, j), \dots, C_i^*(k, j)$ 
3.4 //Imputation  $CIMV_i(i, j)$ 
3.4.1 SWITCH (type of  $j$ )
3.4.1.1 CASE Discrete:  $\hat{M}V_k(i, j) = \text{Mode}\{\text{attribute } j \text{ in } X_C\};$ 
3.4.1.2 CASE Symbol:  $\hat{M}V_k(i, j) = \text{Mode}\{\text{attribute } j \text{ in } X_C\};$ 
3.4.1.3 CASE Continuous:  $\hat{M}V_k(i, j) = \text{Mean}\{\text{attribute } j \text{ in } X_C\};$ 
3.5  $t \leftarrow t + 1$ ; // t is the iteration time
3.6 UNTIL (convergence or  $t \leq 0$ )

```

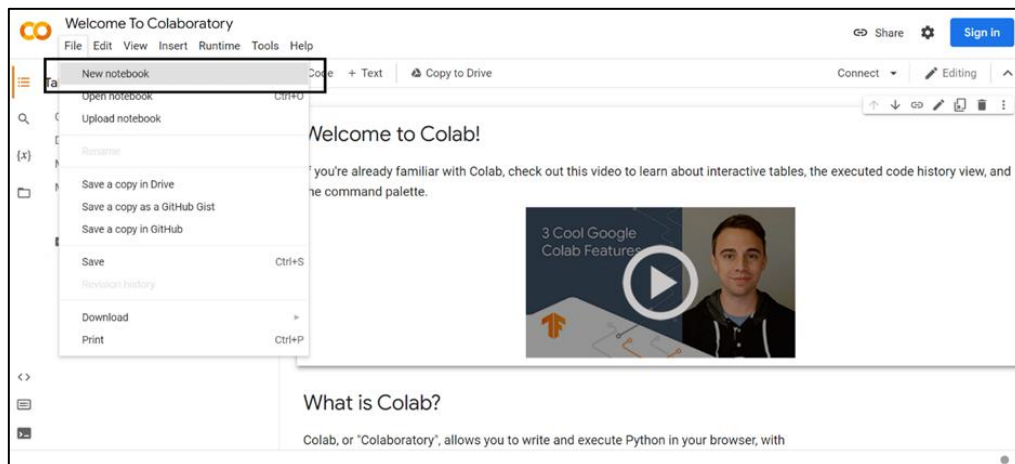

4. Implementation

Below are the steps to run this project code on a different system (Google Colab).

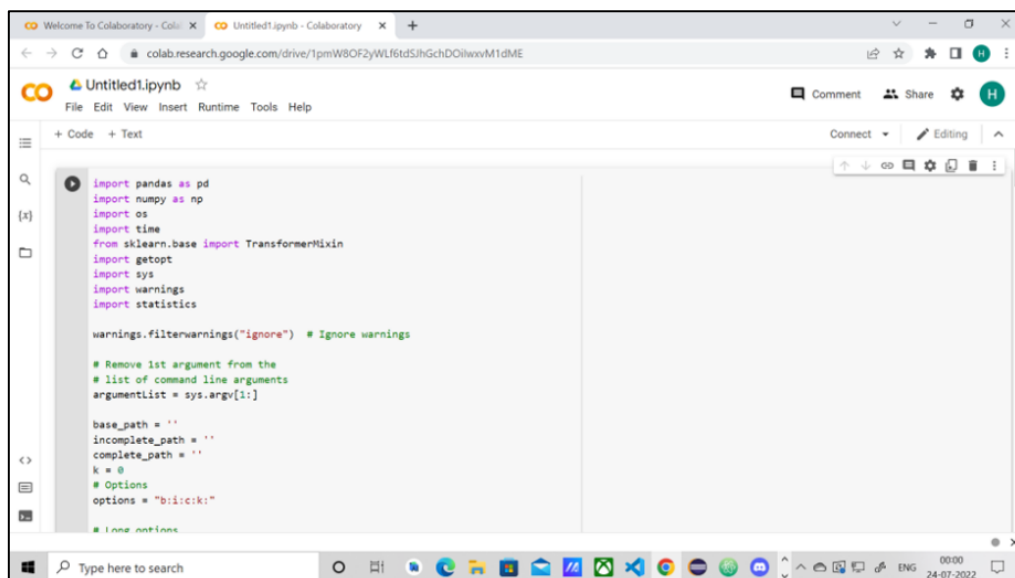
Prerequisite: - You should be logged into your google account.

Step 1: Click on [Google colab url](#) or enter link in browser of your choice.

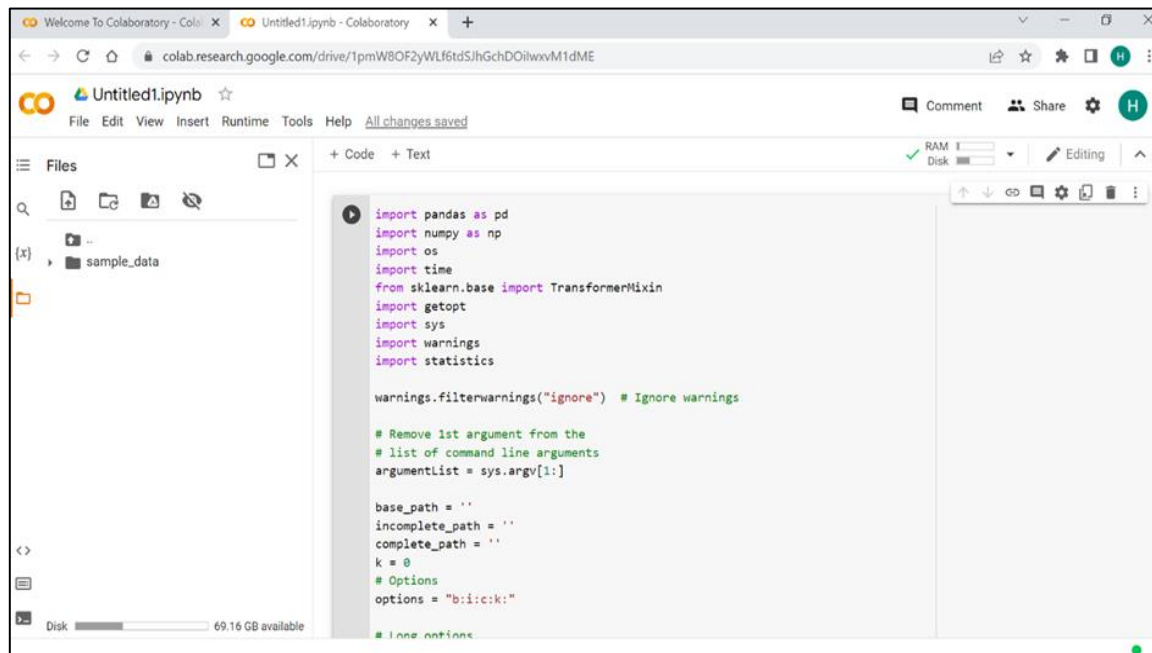
Step 2: Click on File → New Notebook as shown in below screenshot.



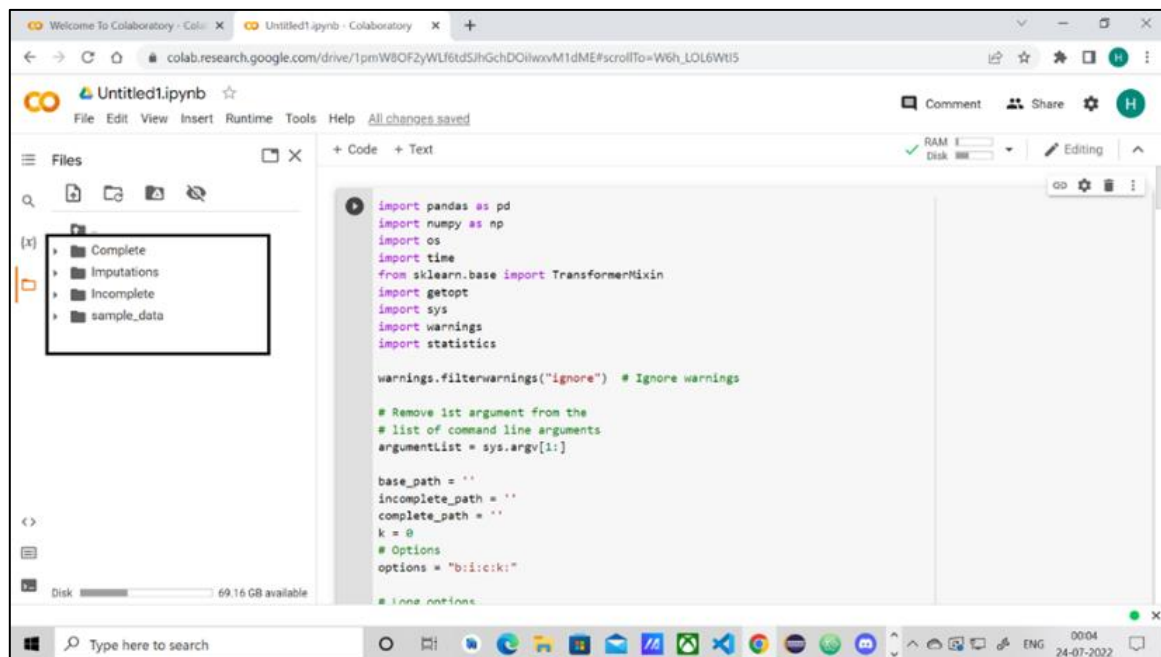
Step 3: Copy Paste the code from main.py into notebook opened in new tab.



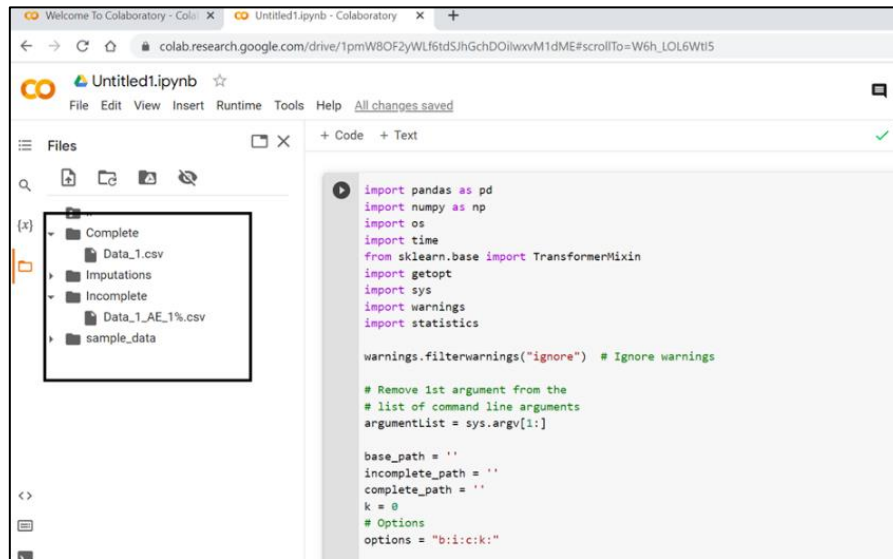
Step 4: Click on the folder icon on the left to upload required files.



Step 5: Right click on space below sample data and create three directories Incomplete, Complete and Imputations.



Step 6: Right click on respective folders and upload incomplete & complete file from data set you wish to impute.

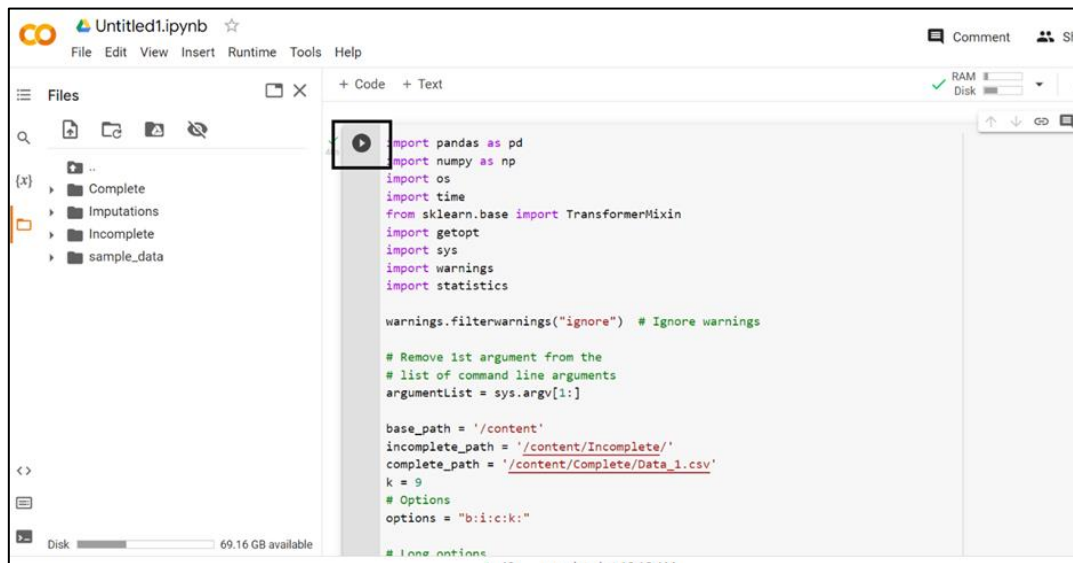


Step 7: update base_path, incomplete_path, complete_path & k with values relevant to your imputations. Considering above example of Data_1 these variables will be.

- base_path = '/content/'
- incomplete_path = '/content/Incomplete/'
- complete_path = '/content/Complete/Data_1.csv'
- k = 9 (nearest neighbors).



Step 8: Click on run cell to execute code.



```
import pandas as pd
import numpy as np
import os
import time
from sklearn.base import TransformerMixin
import getopt
import sys
import warnings
import statistics

warnings.filterwarnings("ignore") # Ignore warnings

# Remove 1st argument from the
# list of command line arguments
argumentList = sys.argv[1:]

base_path = '/content'
incomplete_path = '/content/Incomplete/'
complete_path = '/content/Complete/Data_1.csv'
k = 9
# Options
options = "b:i:c:k:"

# Line options
```

Step 9: Output can be verified by checking imputed file (available in Imputations folder) and console which shows NRMS value and run time.



```
nrmsr = nrms(complete_num_df, imputed_num_df)
print('NRMS :: ',nrmsr)
nrms_data.append([incomplete_df_name_list[index]: nrmsr])
## Compute AE
imputed_ae = 0
for x in range(len(cat_empty_loc[0]) - 1):
    if(imputed_cat_df.iloc[cat_empty_loc[0][x], cat_empty_loc[1][x]] == complete_cat_df.iloc[cat_empty_loc[0][x], cat_empty_loc[1][x]]):
        imputed_ae+=1
AE = (complete_cat_df.size - len(cat_empty_loc[0]) + imputed_ae) / complete_cat_df.size
#ae_data.update( {incomplete_df_name_list[index]: AE} )
#imputed_df_list.append(imputed_df)
imputed_df.to_csv(base_path + '/Imputations/' + incomplete_df_name_list[index] + "_imputed.csv") #save impute
index+=1

print(nrms_data)
#print(ae_data)
```

```
option -f not recognized
Imputing File... Data_1_AE_1%
Starting Imputation...
Ended Imputation...
Total time taken is: 0.0 mins and 48.16136384010315 secs
NRMS :: 0.08731230472100532
[{'Data_1_AE_1%': 0.08731230472100532}]
```

Software packages and programming language used are detailed below

Programming Language: - Python 3.8 or above.

Dependencies/imports

- Pandas 1.4.3
- Numpy 1.23.1
- Sklearn 1.1.1
- OS (Built-in module)
- Time (Built-in module)
- Getopt (Built-in module)
- Sys (Built-in module)
- Warnings (Built-in module)

Recommended platform to run this project: - IntelliJ IDE.

5. Experimental Results

There are two main topics that are introspected in this section. These are, namely: -

- a) Experimental Settings
- b) Results Analysis

Experimental Settings: There are two significant parameters in the research paper that require tuning. The first one being **k** (closest neighbours) and the second one being **max_iter** (the maximum number of iterations before the algorithm converges). These are parameters which are manually defined by the user and require a great deal of effort to lock these values in. Choosing the value of **k** has been an age-old issue and continues to be so. As per the reference paper on which this project is based out of, it is said that an optimal value of a $\sqrt{\text{(number of instances)}}$ would give optimal results. However, things get quite complicated when the size of the data becomes extremely large. When the data becomes large, a large value of **k** would result in a significant complexity in sorting the large dataset. This considerably increases the run time. However, a very small value of **k** would result in overfitting, and thereby the algorithm not generalizing well to previously unseen values. Therefore, a great deal of effort has gone in to choose the ideal value for **k**. The process of tuning **k** started with choosing $k=31$, as $\sqrt{1000}$ is 31. The first dataset had 1000 instances. However, as the datasets started getting larger, the imputations were taking an unreasonably long time., Therefore, **k** was then changed to 20, to reduce the imputation time. However, it was still observed that the imputations were taking a very long time. A further reduction of **k** to 15 saw the imputation time come down significantly. All this time, the NRMS for a given dataset was quite stable, which further vindicated the decision to reduce **k**, and in turn save imputation time. The value of **k** was further reduced to 10, and the imputation time came down even further. Finally, a decision was made to stop **k** at a value of 9. The reason behind this was for extremely small values of **k**, an incoming instance with a missing value would be influenced by a very small set of samples. In other words, it would lead to overfitting. However, for some datasets which were notoriously time consuming to impute, **k** values of 1 and 2 have been assumed as well. With these extremely low values of **k**, the NRMS values increased, which was only to be expected. However, the NRMS increase was not large enough to dissuade us from making the change.

The **max_iter** is another parameter which had to be tuned to ascertain the number of times the algorithm would loop until convergence. This, however, was not as big a challenge, as one of the strengths of Gray KNN is its ability to converge quickly. In the code, there are 2 conditions kept for checking convergence: a) If the average of the imputed values in the present and current iteration is lesser than 10^{-4} . b) If the code loops longer than the **max_iter** value specified. In all the datasets imputed, it has been observed that convergence has been achieved much before the **max_iter** value has been breached. Thus, a meagre value of 2 for **max_iter** has been assumed. Absolutely no change in NRMS has been observed when large values of **max_iter** have been assumed. Therefore, it is evident that convergence is being reached at a very early stage. However, to speed up the imputation process for a few complex and large datasets, a **max_iter** value of 1 has been chosen. Subsequently, an increase in NRMS was observed, but it was small enough to reap the benefits of faster imputation.

Results Analysis: A significant variation has been observed in terms of NRMS and imputation time for the different datasets, owing to the variation in the pattern of missing data, the proportion of missing values, and most importantly the size of the data in terms of instances and features. As expected, it was observed that the 1% dataset performed best in terms of NRMS, followed by the 5% data, the 10% missing data, and finally the 20% missing values data. The NRMS values with the above parameter setting hovered around 0.1 for the 1% missing values data and went up to 0.6 for 20% missing values data.

The NRMS of course was a function of the number of instances and attributes present in the dataset. The Data 1 dataset which is the smallest (1000 rows and 10 columns) expectedly performs the best in terms of NRMS. The bigger datasets like the Data 10 dataset (5000 rows and 50 columns) sees a miniscule degradation in terms of NRMS performance, although the degradation is not conspicuous in terms of the 1% datasets for any of the datasets tested in the project. In cases where the parameters k and max_iters have been varied to faster, a very small degradation in terms of NRMS has been observed. The upshot, however, was the faster imputation times.

For all the datasets imputed in this project, the maximum NRMS value that has been observed has not been greater than 0.6. In terms of imputation times, an expected pattern emerges, with the 1% dataset being the quickest to get imputed, followed by the 5%, 10%, and then the 20% dataset. For smaller datasets like Data 1, the 1% dataset gets imputed in a matter of seconds, whereas the 20% dataset took around 12-13 minutes for imputation. However, for larger datasets with more complex missing patterns, the imputation time increases exponentially with some files taking hours to impute.

This is only to be expected, as the algorithm's complexity analysis has an exponential term for the number of instances, and a linearly increasing term for the number of attributes. To reduce the imputation time, parameter values k and max_iters were altered. The value of k and max_iters were set to 1 for extremely large datasets such as Data 10, Data 11, etc. Below is the boxplot showing the variation in imputation time for datasets with different percentage of missing values.

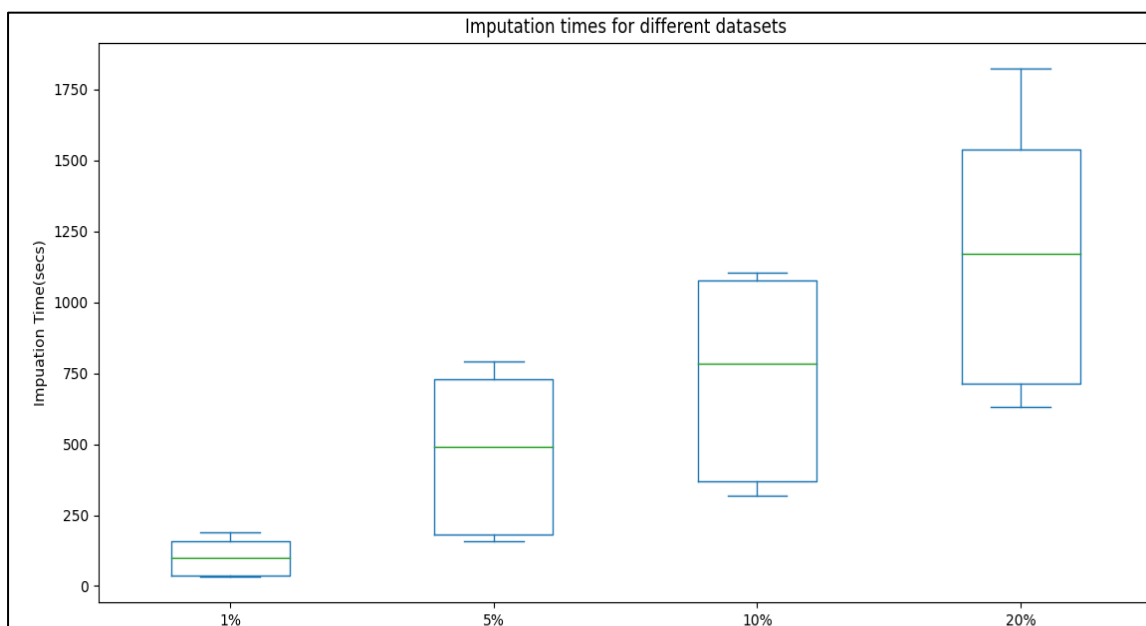


Fig. 1 Boxplot showing the variation in imputation times for different ratios of missing data

From Fig.1 we see that datasets with 1% missing data have the least imputation time. Conversely, datasets with 20% missing data have the highest imputation time. We also see the variance of imputation time increases with an increase in the missing data percentage.

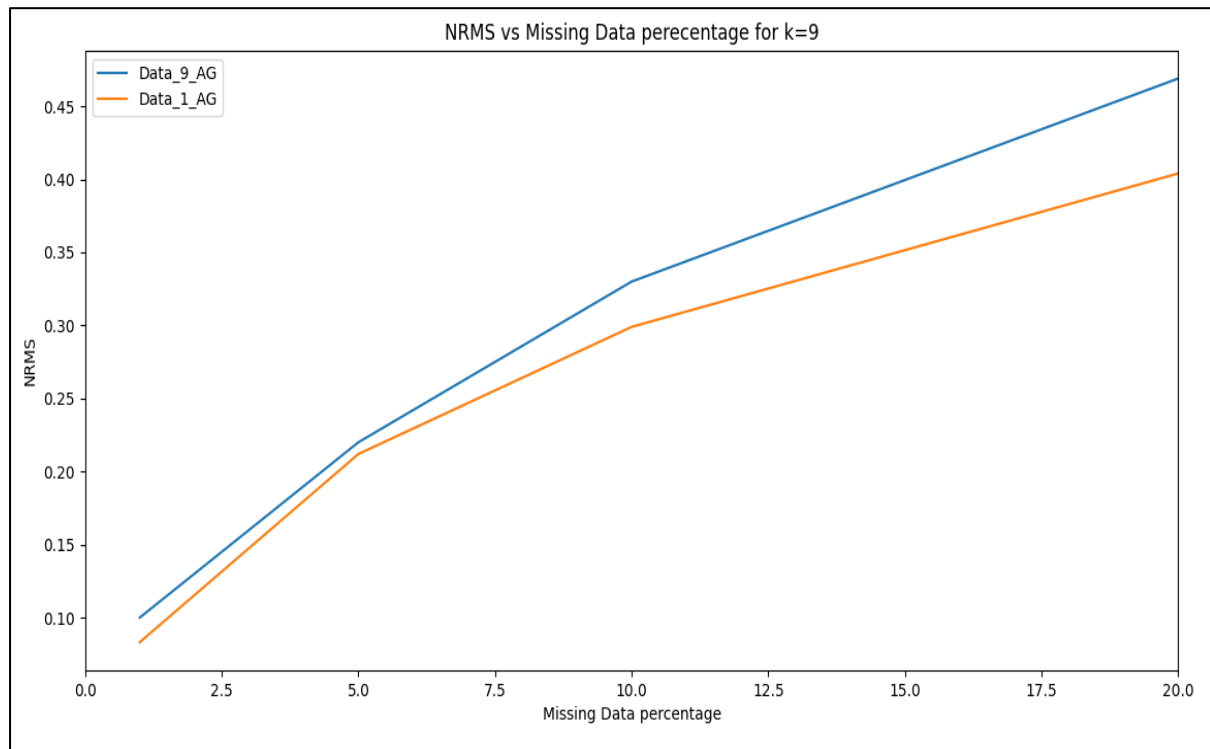


Fig. 2 Comparison of NRMS with different ratios of missing data (1%, 5%, 10%, 20%).

From Fig. 2, we can see how NRMS varies with different ratios of missing data. As expected, 1% missing data has the lowest NRMS data, i.e., around 0.1, and the 20% missing data has the highest NRMS, i.e., around 0.45. Two datasets have been taken to reinforce this claim. Data 1 is a smaller dataset, and hence it has a comparatively low NRMS compared to Data 9. In both cases, we have chosen k as 9.

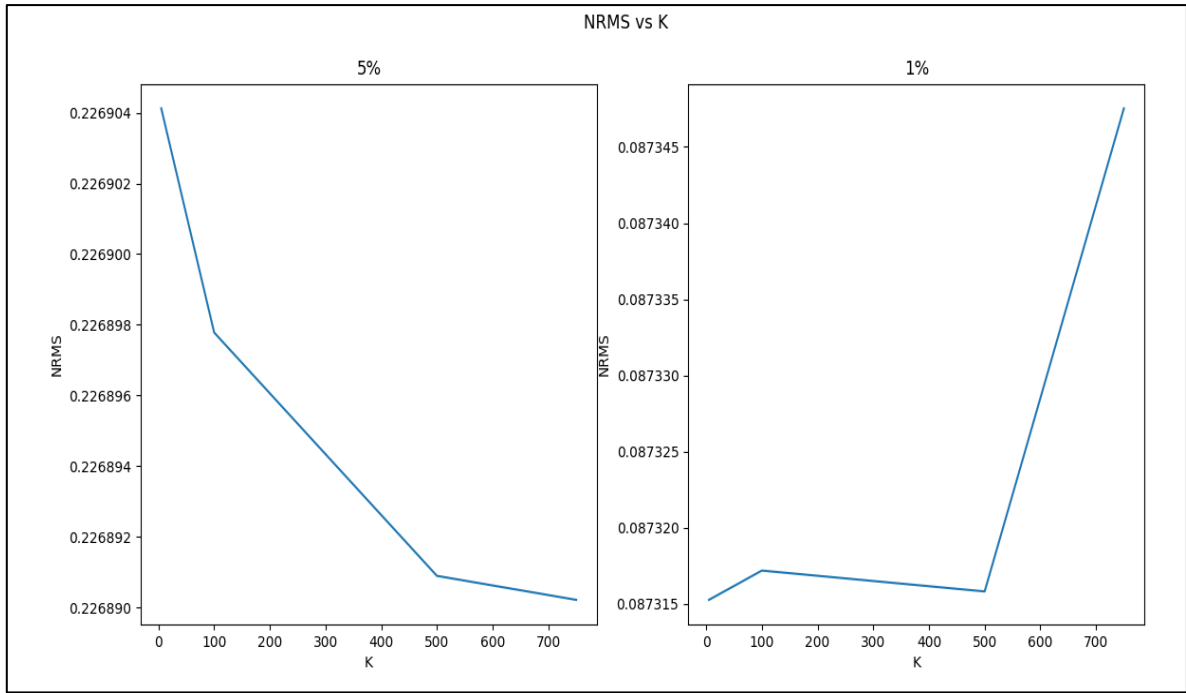


Fig. 3 Comparison of NRMS with different values of k for datasets with 5% and 1% missing values.

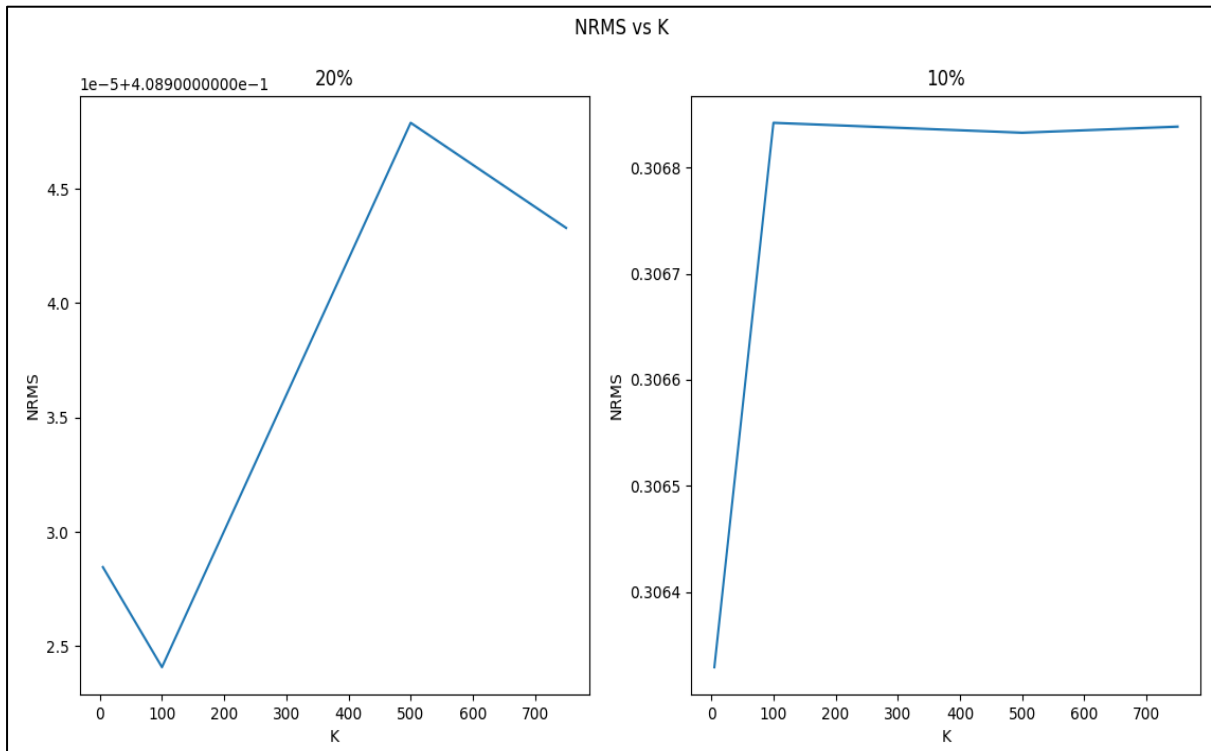


Fig. 4 Comparison of NRMS with different values of k datasets with 20% and 10% missing values.

Fig. 3 and Fig. 4 depict the variation in NRMS for different values of k. Fig. 3 shows the comparison of NRMS with different values of k for datasets which have a missing value ratio of 1% and 5%. Similarly, Fig. 4 shows the comparison of NRMS with different values of k for datasets which have missing value ratios of 20% and 10%. Four values of k have been tested for each case. The values of k that have been used for visualization are k=5, 100, 500 and 750. From the line plots we see a small variation in NRMS when k is being varied.

6. References

- [1] Chen, J., Shao, J., 2000. Nearest neighbor imputation for survey data. *Journal of Official Statistics* 16, 113–132.
- [2] de Andrade Silva Jonathan, Hruschka Eduardo, R., 2009. EACImpute: an evolutionary algorithm for clustering-based imputation. *ISDA 2009*, 1400–1406.
- [3] Lall, U., Sharma, A., 1996. “A nearest-neighbor bootstrap for resampling hydrologic time series”. *Water Resources Research* 32 (3), 679–693.
- [4] Deng, J.L., 1982. Control problems of grey system. *System and Control Letters* 1, 288–294.
- [5] J. Deng, “Introduction to grey system theory,” *The Journal of Grey System*, vol. 1, pp. 1–24, 1989.
- J. Deng, “Grey information space,” *The Journal of Grey System*, vol. 1, pp. 103–117, 1989.
- [6] Jou, J.M., et al., 1999. “The gray prediction search algorithm for block motion estimation”. *IEEE Transactions on Circuits and Systems for Video Technology* 9 (6), 843–848.
- [7] Dan Newton. “Learning Big O Notation with $O(n)$ ”. <https://dzone.com/articles/learning-big-o-notation-with-on-complexity>
- [8] Programs Wiki. “The Programs Encyclopedia”. <https://programs.wiki/wiki/merge-sort-and-heap-sort.html>
- [9] Shichao Zang. “Nearest neighbor selection for iteratively KNN imputation”. *The Journal of Systems and Software*