

Assignment On
“Missing Value Handling”

Course Title: Data Mining and Data Warehousing Lab

Course Code: CSEL-4108

Submitted To,
Dr. Md. Manowarul Islam
Associate Professor
Department of CSE
Jagannath University, Dhaka

Submitted By,
Md. Abu Yousuf Siam
(B180305012)
&
Momtaj Hossain Mow
(B180305025)



Department of Computer Science & Engineering
Jagannath University, Dhaka
Date of Submission: 23/06/2023

Missing Value Handling

Real-world data often has missing values. Data can have missing values for a number of reasons such as observations that were not recorded and data corruption. Handling missing data is important as many machine learning algorithms do not support data with missing values.

1. Diabetes Dataset

The Diabetes Dataset involves predicting the onset of diabetes within 5 years in given medical details.

Raw files needed for this experiment:

- a. Dataset Files¹
- b. Dataset Details²

It is a binary (2-class) classification problem. The number of observations for each class is not balanced. There are 768 observations with 8 input variables and 1 output variable. The variable names are as follows:

1. Number of times pregnant.
2. Plasma glucose concentration: a 2 hours long oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skinfold thickness (mm).
5. 2-Hours serum insulin (mu U/ml).
6. Body Mass Index (weight in kg/(height in m)²).
7. Diabetes pedigree function.
8. Age (in years).
9. Class variable (0 or 1).

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 65%. Top results achieve a classification accuracy of approximately 77%.

¹ <https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv>

² <https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.names>

A sample of the first 5 rows is listed below:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
...
```

This dataset is known to have missing values. Specifically, there are missing observations for some columns that are marked as a zero value. We can corroborate³ this by the definition of those columns and the domain knowledge that a zero value is invalid for those measures, e.g. a zero for body mass index or blood pressure is invalid.

The dataset is given below, it should be renamed to and saved to the current working directory with the file name `pima-indians-diabetes.csv`

File: [pima-indians-diabetes.csv](#)⁴

2. Mark Missing Values

Most data has missing values, and the likelihood of having missing values increases with the size of the dataset.

Missing data are not rare in real data sets. In fact, the chance that at least one data point is missing increases as the data set size increases.

— Page 187, *Feature Engineering and Selection*, 2019.

In this section, we will look at how we can identify and mark values as missing. We can use plots and summary statistics to help identify missing or corrupt data. We can load the dataset as a Pandas DataFrame and print summary statistics on each attribute.

³ confirm or give support to (a statement, theory, or finding).

"the witness had corroborated the boy's account of the attack"

⁴ <https://pastebin.ubuntu.com/p/F9pPn46BJQ/>

Code:

```
# load and summarize the dataset
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the dataset
print(dataset.describe())
```

Running this example produces the following output:

	0	1	2	3	4	5 \
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000

	6	7	8
count	768.000000	768.000000	768.000000
mean	0.471876	33.240885	0.348958
std	0.331329	11.760232	0.476951
min	0.078000	21.000000	0.000000
25%	0.243750	24.000000	0.000000
50%	0.372500	29.000000	0.000000
75%	0.626250	41.000000	1.000000
max	2.420000	81.000000	1.000000

We can see that there are columns that have a minimum value of zero (0). On some columns, a value of zero does not make sense and indicates an invalid or missing value.

⁵Missing values are frequently indicated by out-of-range entries; perhaps a negative number (e.g., -1) in a numeric field that is normally only positive, or a 0 in a numeric field that can never normally be 0.

⁵ -Page 62, Data Mining: Practical Machine Learning Tools and Techniques, 2016

Specifically, the following columns have an invalid zero minimum value:

- 1: Plasma glucose concentration
- 2: Diastolic blood pressure
- 3: Triceps skinfold thickness
- 4: 2-Hour serum insulin
- 5: Body mass index

Let's confirm this by looking at the raw data, the example prints the first 20 rows of data.

```
# load the dataset and review rows
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# print the first 20 rows of data
print(dataset.head(20))
```

Running the example, we can clearly see 0 values in the columns 2, 3, 4, and 5.

	0	1	2	3	4	5	6	7	8
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1
10	4	110	92	0	0	37.6	0.191	30	0
11	10	168	74	0	0	38.0	0.537	34	1
12	10	139	80	0	0	27.1	1.441	57	0
13	1	189	60	23	846	30.1	0.398	59	1
14	5	166	72	19	175	25.8	0.587	51	1
15	7	100	0	0	0	30.0	0.484	32	1
16	0	118	84	47	230	45.8	0.551	31	1
17	7	107	74	0	0	29.6	0.254	31	1
18	1	103	30	38	83	43.3	0.183	33	0
19	1	115	70	30	96	34.6	0.529	32	1

We can get a count of the number of missing values on each of these columns. We can do this by marking all of the values in the subset of the DataFrame we are interested in that have zero values as True. We can then count the number of true values in each column.

We can do this by marking all of the values in the subset of the DataFrame we are interested in that have zero values as True. We can then count the number of true values in each column.

```
# example of summarizing the number of missing values for each
variable
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# count the number of missing values for each column
num_missing = (dataset[[1,2,3,4,5]] == 0).sum()
# report the results
print(num_missing)
```

Running the example prints the following output:

```
1      5
2     35
3    227
4    374
5     11
dtype: int64
```

We can see that columns 1,2 and 5 have just a few zero values, whereas columns 3 and 4 show a lot more, nearly half of the rows.

This highlights that different “missing value” strategies may be needed for different columns, e.g. to ensure that there are still a sufficient number of records left to train a predictive model.

When a predictor is discrete in nature, missingness can be directly encoded into the predictor as if it were a naturally occurring category. ⁶

⁶ — Page 197, [Feature Engineering and Selection](#), 2019.

In Python, specifically Pandas, NumPy and Scikit-Learn, we mark missing values as NaN. Values with a NaN value are ignored from operations like sum, count, etc.

We can mark values as NaN easily with the Pandas DataFrame by using the `replace()` function on a subset of the columns we are interested in.

After we have marked the missing values, we can use the `isnull()` function to mark all of the NaN values in the dataset as True and get a count of the missing values for each column.

```
# example of marking missing values with nan values
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# count the number of nan values in each column
print(dataset.isnull().sum())
```

Running the example prints the number of missing values in each column. We can see that the columns 1:5 have the same number of missing values as zero values identified above. This is a sign that we have marked the identified missing values correctly.

We can see that the columns 1 to 5 have the same number of missing values as zero values identified above. This is a sign that we have marked the identified missing values correctly.

```
0      0
1       5
2      35
3     227
4     374
5      11
6       0
7       0
8       0
dtype: int64
```

This is a useful summary.

Below is the same example, except we print the first 20 rows of data.

```
# example of review rows from the dataset with missing values
marked
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# print the first 20 rows of data
print(dataset.head(20))
```

Running the example, we can clearly see NaN values in the columns 2, 3, 4 and 5. There are only 5 missing values in column 1, so it is not surprising we did not see an example in the first 20 rows.

It is clear from the raw data that marking the missing values had the intended effect.

	0	1	2	3	4	5	6	7	8
0	6	148.0	72.0	35.0	NaN	33.6	0.627	50	1
1	1	85.0	66.0	29.0	NaN	26.6	0.351	31	0
2	8	183.0	64.0	NaN	NaN	23.3	0.672	32	1
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21	0
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33	1
5	5	116.0	74.0	NaN	NaN	25.6	0.201	30	0
6	3	78.0	50.0	32.0	88.0	31.0	0.248	26	1
7	10	115.0	NaN	NaN	NaN	35.3	0.134	29	0
8	2	197.0	70.0	45.0	543.0	30.5	0.158	53	1
9	8	125.0	96.0	NaN	NaN	NaN	0.232	54	1
10	4	110.0	92.0	NaN	NaN	37.6	0.191	30	0
11	10	168.0	74.0	NaN	NaN	38.0	0.537	34	1
12	10	139.0	80.0	NaN	NaN	27.1	1.441	57	0
13	1	189.0	60.0	23.0	846.0	30.1	0.398	59	1
14	5	166.0	72.0	19.0	175.0	25.8	0.587	51	1
15	7	100.0	NaN	NaN	NaN	30.0	0.484	32	1
16	0	118.0	84.0	47.0	230.0	45.8	0.551	31	1
17	7	107.0	74.0	NaN	NaN	29.6	0.254	31	1
18	1	103.0	30.0	38.0	83.0	43.3	0.183	33	0
19	1	115.0	70.0	30.0	96.0	34.6	0.529	32	1

Before we look at handling missing values, let's first demonstrate that having missing values in a dataset can cause problems.

3. Missing Values Causes Problems

Having missing values in a dataset can cause errors with some machine learning algorithms.

Missing values are common occurrences in data. Unfortunately, most predictive modeling techniques cannot handle any missing values. Therefore, this problem must be addressed prior to modeling. — Page 203, Feature Engineering and Selection, 2019.

In this section, we will try to evaluate the **Linear Discriminant Analysis (LDA)** algorithm on the dataset with missing values. This is an algorithm that does not work when there are missing values in the dataset.

The below example marks the missing values in the dataset, as we did in the previous section, then attempts to evaluate LDA using 3-fold cross validation and print the mean accuracy.

```
# example where missing values cause errors
from numpy import nan
from pandas import read_csv
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the model
model = LinearDiscriminantAnalysis()
# define the model evaluation procedure
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
result = cross_val_score(model, X, y, cv=cv,
scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

Running the example results in an error, as follows:

```
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
```

Screenshot of the error:

```
Accuracy: nan

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\model_selection\_validation.py:372: FitFailedWarning:
3 fits failed out of a total of 3.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_score='raise'.

Below are more details about the failures:
-----
3 fits failed with the following error:
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\site-packages\sklearn\model_selection\_validation.py", line 680, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "C:\ProgramData\Anaconda3\lib\site-packages\sklearn\discriminant_analysis.py", line 544, in fit
    X, y = self._validate_data(
  File "C:\ProgramData\Anaconda3\lib\site-packages\sklearn\base.py", line 581, in _validate_data
    X, y = check_X_y(X, y, **check_params)
  File "C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py", line 964, in check_X_y
    X = check_array(
  File "C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py", line 800, in check_array
    _assert_all_finite(array, allow_nan=force_all_finite == "allow-nan")
  File "C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py", line 114, in _assert_all_finite
    raise ValueError(
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').

warnings.warn(some_fits_failed_message, FitFailedWarning)
```

This is as we expect.

We are prevented from evaluating an LDA algorithm (and other algorithms) on the dataset with missing values.

Many popular predictive models such as support vector machines, the glmnet, and neural networks, cannot tolerate any amount of missing values.

— Page 195, *Feature Engineering and Selection*, 2019.

Now, we can look at methods to handle the missing values.

4. Remove Rows With Missing Values

The simplest strategy for handling missing data is to remove records that contain a missing value.

The simplest approach for dealing with missing values is to remove entire predictor(s) and/or sample(s) that contain missing values.

— Page 196, *Feature Engineering and Selection*, 2019.

We can do this by creating a new Pandas DataFrame with the rows containing missing values removed.

Pandas provides the `dropna()` function that can be used to drop either columns or rows with missing data. We can use `dropna()` to remove all rows with missing data, as follows:

```
# example of removing rows that contain missing values
from numpy import nan
from pandas import read_csv
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# summarize the shape of the raw data
print(dataset.shape)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# drop rows with missing values
dataset.dropna(inplace=True)
# summarize the shape of the data with missing rows removed
print(dataset.shape)
```

Running this example, we can see that the number of rows has been aggressively cut from 768 in the original dataset to 392 with all rows containing a NaN removed.

```
(768, 9)
(392, 9)
```

We now have a dataset that we could use to evaluate an algorithm sensitive to missing values like LDA.

Code:

```
# evaluate model on data after rows with missing data are
removed
from numpy import nan
from pandas import read_csv
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# replace '0' values with 'nan'
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# drop rows with missing values
dataset.dropna(inplace=True)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the model
model = LinearDiscriminantAnalysis()
# define the model evaluation procedure
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
result = cross_val_score(model, X, y, cv=cv,
scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

Note: The **results may vary** given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The example runs successfully and **prints** the accuracy of the model.

```
Accuracy: 0.781
```

Removing rows with missing values can be too limiting on some predictive modeling problems, an alternative is to impute missing values.

5. Impute Missing Values

Imputing refers to using a model to replace missing values.

... missing data can be imputed. In this case, we can use information in the training set predictors to, in essence, estimate the values of other predictors.

— Page 42, *Applied Predictive Modeling*, 2013.

There are many options we could consider when replacing a missing value, for example:

- A constant value that has meaning within the domain, such as 0, distinct from all other values.
- A value from another randomly selected record.
- A mean, median or mode value for the column.
- A value estimated by another predictive model.

Any imputing performed on the training dataset will have to be performed on new data in the future when predictions are needed from the finalized model. This needs to be taken into consideration when choosing how to impute the missing values. For example, if we choose to impute with mean column values, these mean column values will need to be stored to file for later use on new data that has missing values.

Pandas provides the **fillna()** function for replacing missing values with a specific value. For example, we can use fillna() to replace missing values with the mean value for each column, as follows:

```
# manually impute missing values with numpy
from pandas import read_csv
from numpy import nan
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# mark zero values as missing or NaN
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# fill missing values with mean column values
dataset.fillna(dataset.mean(), inplace=True)
# count the number of NaN values in each column
print(dataset.isnull().sum())
```

Running the example provides a count of the number of missing values in each column, showing zero missing values.

```
0    0
1    0
2    0
3    0
4    0
5    0
6    0
7    0
8    0
dtype: int64
```

The scikit-learn library provides the **SimpleImputer pre-processing** class that can be used to replace missing values.

It is a flexible class that allows to specify the value to replace (it can be something other than NaN) and the technique used to replace it (such as mean, median, or mode). The **SimpleImputer** class operates directly on the NumPy array instead of the DataFrame.

The example below uses the SimpleImputer class to replace missing values with the mean of each column then prints the number of NaN values in the transformed matrix.

```
# example of imputing missing values using scikit-learn
from numpy import nan
from numpy import isnan
from pandas import read_csv
from sklearn.impute import SimpleImputer
# load the dataset
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# mark zero values as missing or NaN
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# retrieve the numpy array
values = dataset.values
# define the imputer
imputer = SimpleImputer(missing_values=nan, strategy='mean')
# transform the dataset
transformed_values = imputer.fit_transform(values)
# count the number of NaN values in each column
print('Missing: %d' % isnan(transformed_values).sum())
```

Running the example shows that all NaN values were imputed successfully.

```
Missing: 0
```

In either case, we can train algorithms sensitive to NaN values in the transformed dataset, such as LDA.

The example below shows the LDA algorithm trained in the SimpleImputer transformed dataset.

We use a Pipeline to define the modeling pipeline, where data is first passed through the imputer transform, then provided to the model.

This ensures that the imputer and model are both fit only on the training dataset and evaluated on the test dataset within each cross-validation fold. This is important to avoid data leakage.

Code:

```
# example of evaluating a model after an imputer transform
from numpy import nan
from pandas import read_csv
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
dataset = read_csv('pima-indians-diabetes.csv', header=None)
# mark zero values as missing or NaN
dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, nan)
# split dataset into inputs and outputs
values = dataset.values
X = values[:,0:8]
y = values[:,8]
# define the imputer
imputer = SimpleImputer(missing_values=nan, strategy='mean')
# define the model
lda = LinearDiscriminantAnalysis()
# define the modeling pipeline
pipeline = Pipeline(steps=[('imputer', imputer), ('model',
lda)])
# define the cross validation procedure
kfold = KFold(n_splits=3, shuffle=True, random_state=1)
# evaluate the model
result = cross_val_score(pipeline, X, y, cv=kfold,
scoring='accuracy')
# report the mean performance
print('Accuracy: %.3f' % result.mean())
```

Note: The **results may vary** given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example prints the accuracy of LDA on the transformed dataset.

```
Accuracy: 0.762
```

Next we will look at using algorithms that treat missing values as just another value when modeling.

6. Algorithms that Support Missing Values

Not all algorithms fail when there is missing data.

There are algorithms that can be made robust to missing data, such as k-Nearest Neighbors that can ignore a column from a distance measure when a value is missing. Naive Bayes can also support missing values when making a prediction.

One of the really nice things about Naive Bayes is that missing values are no problem at all.

— Page 100, Data Mining: Practical Machine Learning Tools and Techniques, 2016.

There are also algorithms that can use the missing value as a unique and different value when building the predictive model, such as classification and regression trees.

... a few predictive models, especially tree-based techniques, can specifically account for missing data.

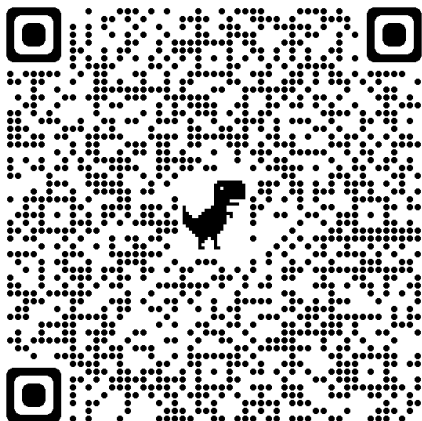
— Page 42, Applied Predictive Modeling, 2013.

Sadly, the scikit-learn implementations of naive bayes, decision trees and k-Nearest Neighbors are *not robust* to missing values. Although it is being considered.

Files: **CSV File** and **Missing Value Handling ipynb file**

Link to this tutorial is **here**.

QR Code to find this tutorial:



Dealing with Missing Values in Python

Data cleaning is one of the most crucial steps for machine and deep learning models to perform well. It involves transforming raw data into a format that the end-user can interpret by handling missing values, removing special characters, handling skewed data, and so on. This article will look into data cleaning and handling missing values.

Generally, missing values are denoted by NaN, null, or None. The dataset's data structure can be improved by removing errors, duplication, corrupted items, and other issues.

Significance of handling the missing values

Effective data management necessitates the ability to fill in blanks. It's a big deal in data analysis because it has such an impact on the outcome.

The results of models with many data gaps are really hard to accept. In a statistical study, skewed estimates could make it unreliable and give people the wrong results.

Problems caused by missing values

- Having missing values makes it more difficult to rule out the **null hypothesis** during testing.
- Parameter estimations could be affected if data is lost.
- The sample's representation may be distorted as a result.
- Because of this, interpreting the study's results may be more difficult.
- The accuracy of models might not be suitable.
- Data inconsistencies might lead to frequent errors while training the model.

Types of missing data types

Several classifications or prediction models depend on the data pattern lacking from the dataset.

a. Missing completely at random (MCAR)

It doesn't matter if there are observed or unobserved data when using MCAR. If data are MCAR, the data can be seen as a simple random sample of the entire dataset of interest.

MCAR is an overly optimistic and frequently unfounded assumption. This assumption occurs when the chance of missing data is unrelated to the prediction value or the observed response to a query.

In simple words, missing data not correlated with the target variable can be ignored.

Solution: Deleting rows or columns.

b. Missing at Random (MAR)

In the case of MAR data, the observed data are systematically linked to the missing data.

A complete case analysis of a data set containing MAR data may or may not result in a bias, depending on whether all relevant data is present and no fields are missing. As long as you consider the known factors, you can objectively analyze the case.

Rather than taking into account a single missing value, a cluster of observed responses has a more significant impact on the likelihood that an experimenter will receive an absent answer.

Solution: Imputation of data.

We attribute the missing data when we find that missing data has a high correlation to the target variable, resulting in better model results.

c. Missing not at Random (MNAR)

When data are MNAR, the missing data is always linked to the unobserved data, which means the missing data is linked to things or events that the researcher can't measure.

Complete case analysis of a data set with MNAR data can be biased because the missing data sources aren't counted. This means that this issue can't be addressed in the analysis, which means that this fact will skew your conclusion about the effect of the data set.

Missing not at random is the only information that is lacking, other than the previously listed categories.

Managing the MNAR datasets is a significant annoyance. Modeling the missing data is the only way to approximate the parameters in this scenario.

Solution: Improve dataset find data.

[For more: **missing data types**]

Types of imputed information

A variety of sizes and shapes are offered in the form of imputations.

To build an accurate model of our application, we must first fill in any data gaps in our dataset.

These are a few techniques:

- **Single Imputation:** Only add missing values to the dataset once, to create an imputed dataset.
- **Univariate Imputation:** This is the case in which only the target variable is used to generate the imputed values.
- **Numerous imputations:** Duplicate missing value imputation across multiple rows of data. To get multiple imputed datasets, you must repeat a single imputation process.
- **Multivariate Imputation:** Impute values based on other variables, such as estimating missing values using linear regression.

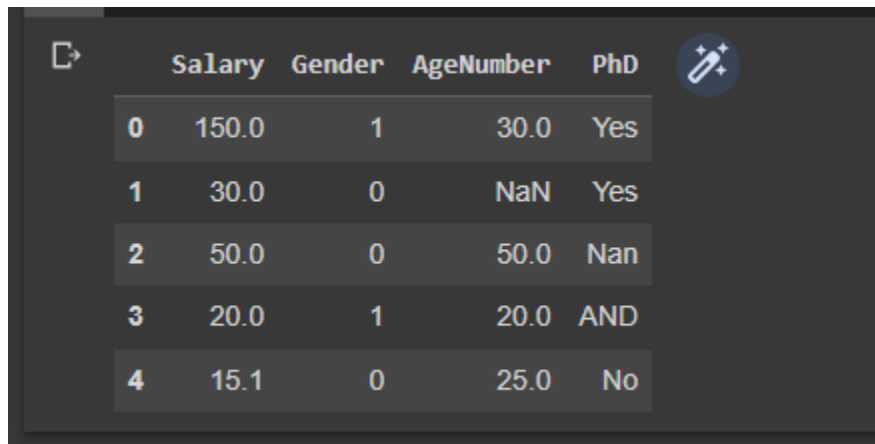
How to fix dataset's missing data

Important: The CSV file for the dataset is attached here: **ipynb file**

There are a variety of approaches to deal with missing data. We will look at some of them, but first, we will start with things like importing libraries.

```
import pandas as pan
import numpy as num
dataset = pan.read_csv("IncomeAndGender.csv")
dataset.head()
```

Output:



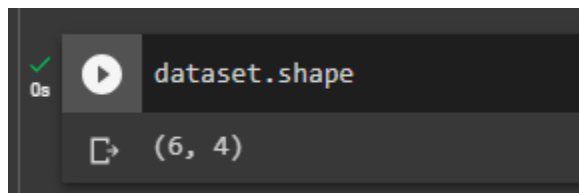
	Salary	Gender	AgeNumber	PhD
0	150.0	1	30.0	Yes
1	30.0	0	NaN	Yes
2	50.0	0	50.0	Nan
3	20.0	1	20.0	AND
4	15.1	0	25.0	No

The CSV file for the dataset is attached here: **ipynb file**

Looking at the dataset's dimensions as a measure of its size:

```
print(dataset.shape)
```

Output:

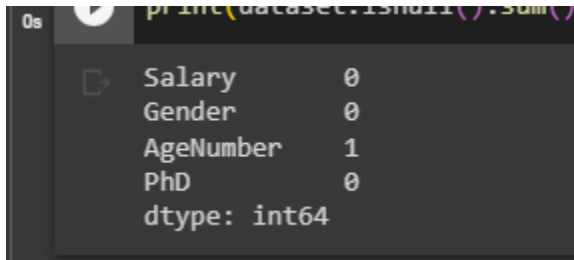


0s	dataset.shape
	(6, 4)

In search for missing information we write this code portion:

```
print(dataset.isnull().sum())
```

Output:



```
Salary      0
Gender      0
AgeNumber   1
PhD         0
dtype: int64
```

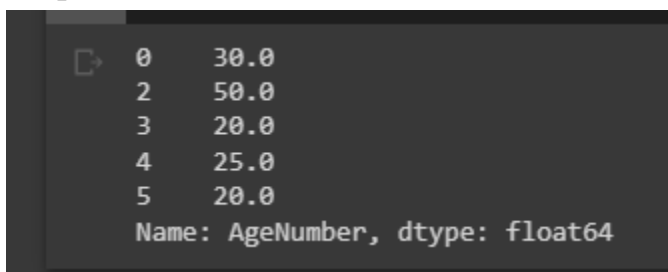
There is nothing to worry about not having enough information. The algorithm decides how to read the data that is given and how it will be used if there isn't enough.

Loss-reduction algorithms can be trained to find the best values for missing data. An error can be made in linear regression. We need to deal with the lack of data until we figure out what went wrong with the model. If it's positive, we'll go ahead. If not, we'll stop.

Code:

```
dataset["AgeNumber"][:10]
```

Output:



```
0    30.0
2    50.0
3    20.0
4    25.0
5    20.0
Name: AgeNumber, dtype: float64
```

Removing the rows/columns that are not in use

The next most straightforward thing to do is leave out observations that don't have any data.

Python's **pandas** module has a method called `dropna()` that can get rid of empty rows. When dealing with machine learning problems, there's no need to fill in every blank in every column.

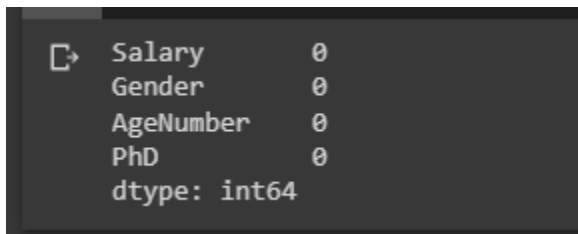
There are both advantages and disadvantages to removing the rows/columns:

- It's Faster and Easier
- Approximately 40% of the data is lost.

Code:

```
dataset.dropna(inplace=True)
print(dataset.isnull().sum())
```

Output:

A terminal window showing the output of the `print(dataset.isnull().sum())` command. The output lists four columns: Salary, Gender, AgeNumber, and PhD, each with a value of 0. Below these, it shows `dtype: int64`.

```
Salary      0
Gender      0
AgeNumber   0
PhD         0
dtype: int64
```

Imputation based on the mean

Each missing value can be restored after calculating the non-missing values in a column. Using this method with anything other than numbers is severely restricted. It's a simple way to analyze small amounts of data. One flaw is the lack of feature correlations, but there are others. This technique only works with one column at a time. A skewed mean value will likely replace an outlier treatment.

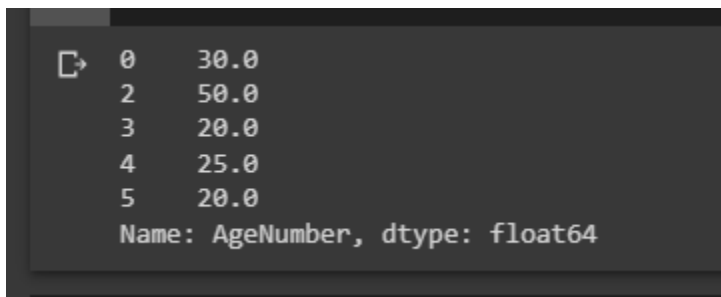
The technique only works with numerical datasets and fails when independent variables are correlated.

Using the mean also destroys the relationships between variables. True, the inserted mean preserves the observed data mean. Even when data are missing at random, a fair and accurate mean estimate can be obtained:

Code:

```
dataset["AgeNumber"] = dataset["AgeNumber"].replace(num.NaN,  
dataset["AgeNumber"].mean())  
print(dataset["AgeNumber"][:10])
```

Output:



```
0    30.0  
2    50.0  
3    20.0  
4    25.0  
5    20.0  
Name: AgeNumber, dtype: float64
```

Using the median to compute

Using median values is another method of Imputation that addresses the previous method's outlier issue.

An outlier is an object or data item significantly different from the rest of the dataset.

When sorting, a column's center value is updated rather than an outlier. No correlation between the independent variables was found, and it only works with numerical datasets. An independent variable is what a user changes precisely.

Code:

```
dataset["AgeNumber"] = dataset["AgeNumber"].replace(num.NaN,  
dataset["AgeNumber"].median())  
print(dataset["AgeNumber"][:10])
```


Output:

```
0    30.0
2    50.0
3    20.0
4    25.0
5    20.0
Name: AgeNumber, dtype: float64
```

Imputation based on the most common values (mode)

It can be applied to categorical variables with a restricted number of values.

Education level is an excellent example of an ordinal absolute attribute that falls into this category. Since feature relationships are not considered when utilizing this procedure, data bias can occur. If the category values are not evenly distributed among the classes, biasing the data increases.

It is compatible with all data formats, and the value of covariance between independent features cannot be predicted:

Code:

```
import statistics
dataset["AgeNumber"] = dataset["AgeNumber"].replace(num.NaN,
statistics.mode(dataset["AgeNumber"]))
print(dataset["AgeNumber"][:10])
```

Output:

```
0    30.0
2    50.0
3    20.0
4    25.0
5    20.0
Name: AgeNumber, dtype: float64
```

Interpolation–Linear

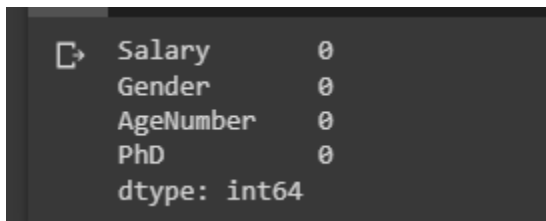
A straight line is used to join dots in increasing order to approximate a missing value.

For the most part, the unknown value is calculated in the same ascending order as the previous values. We don't have to specify Linear Interpolation because it is the default method.

Almost always, it will be used in a time-series dataset.

```
dataset["AgeNumber"] =  
dataset["AgeNumber"].interpolate(method='linear',  
limit_direction='forward', axis=0)  
dataset.isnull().sum()
```

Output:



```
Salary      0  
Gender      0  
AgeNumber   0  
PhD         0  
dtype: int64
```

To fill in the blanks in our dataset, we can use the concepts mentioned earlier.

When it comes to finding missing values, there isn't a single method that works best. Finding missing values differs based on the feature and application we want to use.

As a result, we'll have to experiment to find the best solution for our application.

Conclusion

Data cleaning is a feature of the pre-processing data module that we explored in this post. Furthermore, data loss may lead to skewed parameter estimations, reduced sample representativeness, and more complex research analysis.

In conclusion, we looked at various approaches to handling missing data and how these techniques are used.

This tutorial is from [here](#)

References

- [1] [Causes and solutions of missing data](#)
- [2] [See the whole code of the tutorial](#)
- [3] [Handling missing data](#)

QR Code for this part of the assignment:

