Structured vs unstructured data

Structured data is highly-organized and formatted in a way so it's easily searchable in relational databases. Unstructured data has no pre-defined format or organization, making it much more difficult to collect, process, and analyze.
In addition to being collected, processed, and analyzed in different ways, structured and unstructured data will reside in completely different databases.

Unstructured data

Unstructured data is the data which does not conforms to a data model and has no easily identifiable structure such that it can not be used by a computer program easily. Unstructured data is not organised in a pre-defined manner or does not have a pre-defined data model, thus it is not a good fit for a mainstream relational database.

Characteristics of Unstructured Data:

•Data neither conforms to a data model nor has any structure.
•Data can not be stored in the form of rows and columns as in Databases.
•Data does not follows any semantic or rules.
•Data lacks any particular format or sequence.
•Data has no easily identifiable structure.
•Due to lack of identifiable structure, it can not used by computer programs easily.

Sources of Unstructured Data:

•Web pages
•Images (JPEG, GIF, PNG, etc.)
•Videos
•Memos

Advantages of Unstructured Data:

•Its supports the data which lacks a proper format or sequence
•The data is not constrained by a fixed schema
•Very Flexible due to absence of schema
•Data is portable
•It is very scalable
•It can deal easily with the heterogeneity of sources
•These type of data have a variety of business intelligence and analytics applications

Disadvantages Of Unstructured data:

•It is difficult to store and manage unstructured data due to lack of schema and structure.
•Indexing the data is difficult and error prone due to unclear structure and not having pre-defined attributes. Due to which search results are not very accurate.
•Ensuring security to data is difficult task.

Extracting information from unstructured Data:

Unstructured data do not have any structure. So it can not easily interpreted by conventional algorithms. It is also difficult to tag and index unstructured data. So extracting information from them is tough job. Here are possible solutions:

•Taxonomies or classification of data helps in organising data in hierarchical structure. Which will make search process easy.
•Data can be stored in virtual repository and be automatically tagged. For example Documentum.
•Use of application platforms like XOLAP.
XOLAP helps in extracting information from e-mails and XML based documents .
•Use of various data mining tools

Structured data

Structured data is the data which conforms to a data model, has a well define structure, follows a consistent order and can be easily accessed and used by a person or a computer program.

Characteristics of Structured Data:

•Data conforms to a data model and has easily identifiable structure
•Data is stored in the form of rows and columns
Example : Database
•Data is well organised so, Definition, Format and Meaning of data is explicitly known
•Data resides in fixed fields within a record or file
•Similar entities are grouped together to form relations or classes
•Entities in the same group have same attributes
•Easy to access and query, So data can be easily used by other programs
•Data elements are addressable, so efficient to analyse and process

Sources of Structured Data:

•SQL Databases
•Spreadsheets such as Excel
•OLTP Systems
•Online forms
•Sensors such as GPS or RFID tags
•Network and Web server logs
•Medical devices

Advantages of Structured Data:

•Structured data have a well defined structure that helps in easy storage and access of data
•Data can be indexed based on text string as well as attributes. This makes search operation hassle-free
•Data mining is easy i.e knowledge can be easily extracted from data
•Operations such as Updating and deleting is easy due to well structured form of data
•Business Intelligence operations such as Data warehousing can be easily undertaken
•Easily scalable in case there is an increment of data
•Ensuring security to data is easy

Structured data accounts for only about 20% of data but because of its high degree of organisation and performance make it foundation of Big data.

Data Cleaning in Pyton

Data cleaning is one of the important parts of machine learning. It plays a significant part in building a model. It surely isn't the fanciest part of machine learning and at the same time, there aren't any hidden tricks or secrets to uncover. However, proper data cleaning can make or break your project. Professional data scientists usually spend a very large portion of their time on this step. Because of the belief that, "Better data beats fancier algorithms". If we have a well-cleaned dataset, we can get desired results even with a very simple algorithm, which can prove very beneficial at times.

Obviously, different types of data will require different types of cleaning. However, this systematic approach can always serve as a good starting point.

Steps involved in Data Cleaning:

```
1. Removal of unwanted observations
2. Fixing Structural errors
3. Managing Unwanted outliers
4. Handling missing data
```

Removal of unwanted observations

This includes deleting duplicate/ redundant or irrelevant values from our dataset. Duplicate observations most frequently arise during data collection and Irrelevant observations are those that don't actually fit the specific problem that we're trying to solve.
•Redundant observations alter the efficiency by a great extent as the data repeats and may add towards the correct side or towards the incorrect side, thereby producing unfaithful results.
•Irrelevant observations are any type of data that is of no use to us and can be removed directly.

Fixing Structural errors

The errors that arise during measurement, transfer of data or other similar situations are called structural errors. Structural errors include typos in the name of features, same attribute with different name, mislabeled classes, i.e. separate classes that should really be the same or inconsistent capitalization.
•For example, the model will treat America and america as different classes or

Managing Unwanted outliers

Outliers can cause problems with certain types of models. For example, linear regression models are less robust to outliers than decision tree models. Generally, we should not remove outliers until we have a legitimate reason to remove them. Sometimes, removing them improves performance, sometimes not. So, one must have a good reason to remove the outlier, such as suspicious measurements that are unlikely to be the part of real data.

Handling missing data

Missing data is a deceptively tricky issue in machine learning. We cannot just ignore or remove the missing observation. They must be handled carefully as they can be an indication of something important. The two most common ways to deal with missing data are:

```
    1. Dropping observations with missing values.
```

Dropping missing values is sub-optimal because when you drop observations, you drop information.
•The fact that the value was missing may be informative in itself.
•Plus, in the real world, we often need to make predictions on new data even if some of the features are missing!

```
    2. Imputing the missing values from past observations.
```

Imputing missing values is sub-optimal because the value was originally missing but we filled it in, which always leads to a loss in information, no matter how sophisticated our imputation method is.
•Again, “missingness” is almost always informative in itself, and we should tell your algorithm if a value was missing.
•Even if we build a model to impute our values, we’re not adding any real information. We’re just reinforcing the patterns already provided by other features.

Both of these approaches are sub-optimal because dropping an observation means dropping information, thereby reducing data and imputing values also is sub-optimal as we fil the values that were not present in the actual dataset, which leads to a loss of information.

Missing data is like missing a puzzle piece. If we drop it, that’s like pretending the puzzle slot isn’t there. If we impute it, that’s like trying to squeeze in a piece from somewhere else in the puzzle. So, missing data is always informative and indication of something important. And we must aware our algorithm of missing

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

We will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here we'll refer to missing data in general as null, NaN, or NA values.

There are a number of schemes that have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a mask that globally indicates missing values, or choosing a sentinel value that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like NaN are not available for all data types.

Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point NaN value, and the Python None object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

None: Pythonic missing data

The first sentinel value used by Pandas is None, a Python singleton object that is often used for missing data in Python code. Because it is a Python object, None cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects):

In [24]:
```
import numpy as np
import pandas as pd
vals1 = np.array([1, None, 3, 4])
vals1
```

Out[24]:
```
array([1, None, 3, 4], dtype=object)
```

This dtype=object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:

The use of Python objects in an array also means that if we perform aggregations like sum() or min() across an array with a None value, we will generally get an error:

In [25]:
```
vals1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent ca
<ipython-input-25-30a3fc8c6726> in <module>
----> 1 vals1.sum()

C:\ProgramData\Anaconda3\lib\site-packages\numpy\core\_methods.py i
     34 def _sum(a, axis=None, dtype=None, out=None, keepdims=False
     35          initial=_NoValue):
---> 36     return umr_sum(a, axis, dtype, out, keepdims, initial)
     37 
     38 def _prod(a, axis=None, dtype=None, out=None, keepdims=Fals

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

This reflects the fact that addition between an integer and None is undefined.

NaN: Missing numerical data

```
In [26]:  vals2 = np.array([1, np.nan, 3, 4])
          vals2.dtype

Out[26]:  dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. We should be aware that NaN is a bit like a data virus–it infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN:

```
In [27]:  1 + np.nan

Out[27]:  nan
```

```
In [28]:  0 *  np.nan

Out[28]:  nan
```

Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
In [29]:  vals2.sum(), vals2.min(), vals2.max()

Out[29]:  (nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

```
In [30]:  np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)

Out[30]:  (8.0, 1.0, 4.0)
```

Keep in mind that NaN is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

In [31]:
```
pd.Series([1, np.nan, 2, None])
```

Out[31]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to np.nan, it will automatically be upcast to a floating-point type to accommodate the NA:

In [32]:
```
x = pd.Series(range(2), dtype=int)
x
```

Out[32]:
```
0    0
1    1
dtype: int32
```

In [33]:
```
x[0] = None
x
```

Out[33]:
```
0    NaN
1    1.0
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the None to a NaN value.

While this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works quite well in practice and only rarely causes issues.

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

Keep in mind that in Pandas, string data is always stored with an object dtype.

Operating on Null Values

As we have seen, Pandas treats None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:
•isnull(): Generate a boolean mask indicating missing values
•notnull(): Opposite of isnull()
•dropna(): Return a filtered version of the data
•fillna(): Return a copy of the data with missing values filled or imputed

In [34]:
```
data = pd.Series([1, np.nan, 'hello', None])
data
```

Out[34]:
```
0        1
1      NaN
2    hello
3     None
dtype: object
```

In [35]:
```
data.isnull()
```

Out[35]:
```
0    False
1     True
2    False
3     True
dtype: bool
```

Boolean masks can be used directly as a Series or DataFrame index:

In [36]:
```
data[data.notnull()]
```

Out[36]:
```
0        1
2    hello
dtype: object
```

The isnull() and notnull() methods produce similar Boolean results for DataFrames.

Dropping null values

In addition to the masking used before, there are the convenience methods, dropna() (which removes NA values) and fillna() (which fills in NA values). For a Series, the result is straightforward:

In [37]:

```
data.dropna()
```

Out[37]:

```
0        1
2    hello
dtype: object
```

For a DataFrame, there are more options. Consider the following DataFrame:

In [38]:

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

Out[38]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

We cannot drop single values from a DataFrame; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so dropna() gives a number of options for a DataFrame.

By default, dropna() will drop all rows in which any null value is present:

In [39]:

```
df.dropna()
```

Out[39]:

	0	1	2
1	2.0	3.0	5

Alternatively, we can drop NA values along a different axis; axis=1 drops all columns containing a null value:

In [40]:

```
df.dropna(axis='columns')
```

Out[40]:

	2
0	2
1	5
2	6

In [41]:

```
df[3] = np.nan
df
```

Out[41]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [42]:

```
df.dropna(axis='columns', how='all')
```

Out[42]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

For finer-grained control, the thresh parameter lets us specify a minimum number of non-null values for the row/column to be kept:

In [43]:

```
df.dropna(axis='rows', thresh=3)
```

Out[43]:

	0	1	2	3
1	2.0	3.0	5	NaN

Here the first and last row have been dropped, because they contain only two non-null values.

Filling null values

Sometimes rather than dropping NA values, we'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. We could do this in-place using the isnull() method as a mask, but because it is such a common operation Pandas provides the fillna() method, which returns a copy of the array with the null values replaced.

Consider the following Series:

```
In [44]:  data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
           data
```

```
Out[44]:  a    1.0
          b    NaN
          c    2.0
          d    NaN
          e    3.0
          dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
In [45]:  data.fillna(0)
```

```
Out[45]:  a    1.0
          b    0.0
          c    2.0
          d    0.0
          e    3.0
          dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
In [46]:  # forward-fill
          data.fillna(method='ffill')
```

```
Out[46]:  a    1.0
          b    1.0
          c    2.0
          d    2.0
          e    3.0
          dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
In [47]:  # back-fill
          data.fillna(method='bfill')
```

```
Out[47]:  a    1.0
          b    2.0
          c    2.0
          d    3.0
          e    3.0
          dtype: float64
```

For DataFrames, the options are similar, but we can also specify an axis along which the fills take place:

In [48]:
```
df
```
Out[48]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [49]:
```
df.fillna(method='ffill', axis=1)
```
Out[49]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Notice that if a previous value is not available during a forward fill, the NA value remains.

Handle Missing Data with Python on Pima Indians Diabetes Dataset

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

It is a binary (2-class) classification problem. 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 after 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skinfold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index (weight in kg/(height in m)^2).
7. Diabetes pedigree function.
8. Age (years).
9. Class variable (0 or 1).

This dataset is known to have missing values.

Specifically, there are missing observations for some columns that are marked as a zero value.

In [50]:
```python
from pandas import read_csv
dataset = read_csv('pima-indians-diabetes.csv', header=None)
dataset.head()
```

Out[50]:

	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

In [51]:
```python
dataset.shape
```

Out[51]:
```
(768, 9)
```

In [52]:
```python
dataset.describe().T
```

Out[52]:

	count	mean	std	min	25%	50%	75%	max
0	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	17.00
1	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	199.00
2	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	122.00
3	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	99.00
4	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	846.00
5	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	67.10
6	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	2.42
7	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	81.00
8	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	1.00

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.

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

In [53]:

```
# print the first 20 rows of data
dataset.head(20)
```

Out[53]:

	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 clearly see 0 values in the columns 2, 3, 4, and 5.
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.

```
In [54]:  (dataset[[1,2,3,4,5]] == 0).sum()
```

```
Out[54]:  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.

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.

```
In [55]:  # mark zero values as missing or NaN
          dataset[[1,2,3,4,5]] = dataset[[1,2,3,4,5]].replace(0, np.NaN)
          # count the number of NaN values in each column
          dataset.isnull().sum()
```

```
Out[55]:  0      0
          1      5
          2     35
          3    227
          4    374
          5     11
          6      0
          7      0
          8      0
          dtype: int64
```

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.

This is a useful summary. Let's look at the actual data though, to confirm that 0 has

```
In [56]: dataset.head(20)
```

Out[56]:

	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

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

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:

In [57]:
```
# drop rows with missing values
dataset.dropna(inplace=True)
# summarize the number of rows and columns in the dataset
dataset.shape
```

Out[57]:
```
(392, 9)
```

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.

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

Impute Missing Values

Imputing refers to using a model to replace missing values.

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 you 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:

```
In [58]:  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, np.NaN)
          # fill missing values with mean column values
          dataset.fillna(dataset.mean(), inplace=True)
          # count the number of NaN values in each column
          dataset.isnull().sum()
```

```
Out[58]:  0    0
          1    0
          2    0
          3    0
          4    0
          5    0
          6    0
          7    0
          8    0
          dtype: int64
```

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

The scikit-learn library provides the Imputer() pre-processing class that can be used to replace missing values.

It is a flexible class that allows you 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 Imputer class operates directly on the NumPy array instead of the DataFrame.

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

In [59]:

```
from sklearn.preprocessing import Imputer
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, np.NaN)
# fill missing values with mean column values
values = dataset.values
imputer = Imputer()
transformed_values = imputer.fit_transform(values)
# count the number of NaN values in each column
np.isnan(transformed_values).sum()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\deprecatio
  warnings.warn(msg, category=DeprecationWarning)
```

Out[59]:

```
0
```

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

Descriptive Statistics in Python

Python Descriptive Statistics process describes the basic features of data in a study. It delivers summaries on the sample and the measures and does not use the data to learn about the population it represents. Under descriptive statistics, fall two sets of properties- central tendency and dispersion. Python Central tendency characterizes one central value for the entire distribution. Measures under this include mean, median, and mode. Python Dispersion is the term for a practice that characterizes how apart the members of the distribution are from the center and from each other. Variance/Standard Deviation is one such measure of variability.

Central Tendency in Python

mean()

This function returns the arithmetic average of the data it operates on. If called on an empty container of data, it raises a StatisticsError.

In [60]:

```
import statistics as st
nums=[1,2,3,5,7,9]
st.mean(nums)
```

Out[60]:

```
4.5
```

```
In [61]:  #Negative numbers
          st.mean([-2,-4,7])

Out[61]:  0.3333333333333333
```

mode()

This function returns the most common value in a set of data. This gives us a great idea of where the center lies.

```
In [62]:  nums=[1,2,3,5,7,9,7,2,7,6]
          st.mode(nums)

Out[62]:  7
```

```
In [63]:  st.mode(['A','B','b','B','A','B'])

Out[63]:  'B'
```

median()

For data of odd length, this returns the middle item; for that of even length, it returns the average of the two middle items.

```
In [64]:  st.median(nums)

Out[64]:  5.5
```

harmonic_mean()

This function returns the harmonic mean of the data. For three values a, b, and c, the harmonic mean is-
3/(1/a + 1/b +1/c)
It is a measure of the center; one such example would be speed.

```
In [65]:  st.harmonic_mean([2,4,9.7])

Out[65]:  3.516616314199396
```

For the same set of data, the arithmetic mean would give us a value:

```
In [67]:  st.mean([2,4,9.7])
Out[67]:  5.233333333333333
```

median_low()

When the data is of an even length, this provides us the low median of the data. Otherwise, it returns the middle value.

```
In [68]:  st.median_low([1,2,4])
Out[68]:  2
```

```
In [69]:  st.median_low([1,2,3,4])
Out[69]:  2
```

median_high()

Like median_low, this returns the high median when the data is of an even length. Otherwise, it returns the middle value.

```
In [70]:  st.median_high([1,2,4])
Out[70]:  2
```

```
In [71]:  st.median_high([1,2,3,4])
Out[71]:  3
```

median_grouped()

This function uses interpolation to return the median of grouped continuous data. This is the 50th percentile.

```
In [72]:  st.median([1,3,3,5,7])
Out[72]:  3
```

```
In [73]:  st.median_grouped([1,3,3,5,7],interval=1)
Out[73]:  3.25
```

```
In [74]:   st.median_grouped([1,3,3,5,7],interval=2)
Out[74]:   3.5
```

Dispersion in Python

variance()

This returns the variance of the sample. This is the second moment about the mean and a larger value denotes a rather spread-out set of data. We can use this when our data is a sample out of a population.

```
In [75]:   st.variance(nums)
Out[75]:   7.433333333333334
```

pvariance()

This returns the population variance of data. Use this to calculate variance from an entire population.

```
In [76]:   st.pvariance(nums)
Out[76]:   6.69
```

stdev()

This returns the standard deviation for the sample. This is equal to the square root of the sample variance.

```
In [77]:   st.stdev(nums)
Out[77]:   2.7264140062238043
```

pstdev()

This returns the population standard deviation. This is the square root of population variance.

In [78]:

```
st.pstdev(nums)
```

Out[78]:

```
2.5865034312755126
```

Descriptive Statistics on Pima Indians Diabetes Dataset

Descriptive statistics is a helpful way to understand characteristics of our data and to get a quick summary of it.

The describe() function on the Pandas DataFrame lists 8 statistical properties of each attribute:
•Count
•Mean
•Standard Devaition
•Minimum Value
•25th Percentile
•50th Percentile (Median)
•75th Percentile
•Maximum Value

In [80]:

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'a
data = read_csv('pima-indians-diabetes.csv', names=names)
pd.set_option('display.width', 100)
pd.set_option('precision', 3)
description = data.describe().T
description
```

Out[80]:

	count	mean	std	min	25%	50%	75%	max
preg	768.0	3.845	3.370	0.000	1.000	3.000	6.000	17.00
plas	768.0	120.895	31.973	0.000	99.000	117.000	140.250	199.00
pres	768.0	69.105	19.356	0.000	62.000	72.000	80.000	122.00
skin	768.0	20.536	15.952	0.000	0.000	23.000	32.000	99.00
test	768.0	79.799	115.244	0.000	0.000	30.500	127.250	846.00
mass	768.0	31.993	7.884	0.000	27.300	32.000	36.600	67.10
pedi	768.0	0.472	0.331	0.078	0.244	0.372	0.626	2.42
age	768.0	33.241	11.760	21.000	24.000	29.000	41.000	81.00
class	768.0	0.349	0.477	0.000	0.000	0.000	1.000	1.00

Now we replace 0 with NaN in columns where 0 is missing value and see how the

```
In [125]:
data = read_csv('pima-indians-diabetes.csv', names = names)
# mark zero values as missing or NaN
data[['plas', 'pres', 'skin', 'test', 'mass']] = data[['plas', 'pre
```

```
In [126]:
data.describe().T
```

Out[126]:

	count	mean	std	min	25%	50%	75%	max
preg	768.0	3.845	3.370	0.000	1.000	3.000	6.000	17.00
plas	763.0	121.687	30.536	44.000	99.000	117.000	141.000	199.00
pres	733.0	72.405	12.382	24.000	64.000	72.000	80.000	122.00
skin	541.0	29.153	10.477	7.000	22.000	29.000	36.000	99.00
test	394.0	155.548	118.776	14.000	76.250	125.000	190.000	846.00
mass	757.0	32.457	6.925	18.200	27.500	32.300	36.600	67.10
pedi	768.0	0.472	0.331	0.078	0.244	0.372	0.626	2.42
age	768.0	33.241	11.760	21.000	24.000	29.000	41.000	81.00
class	768.0	0.349	0.477	0.000	0.000	0.000	1.000	1.00

We can see count has decreased in columns having missing values and also other metrics have also changes as any missing value or NaN value is automatically skipped.

Another useful method is value_counts() which can get count of each category in a categorical attributed series of values. For an instance suppose we are dealing with a dataset of customers who are divided as 0 and 1 categories under column name class. We can run this statement to know how many people fall in respective categories.

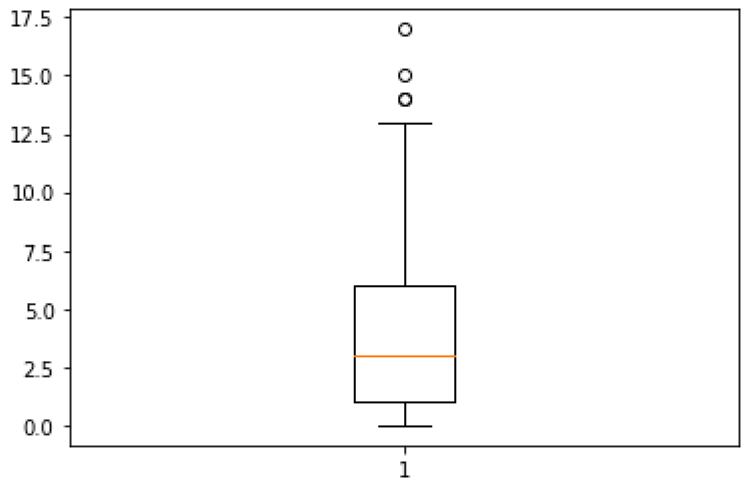
```
In [128]:
data["class"].value_counts()
```

```
Out[128]:
0    500
1    268
Name: class, dtype: int64
```

One more useful tool is boxplot which we can use through matplotlib module. Boxplot is a pictorial representation of distribution of data which shows extreme values, median and quartiles. We can easily figure out outliers by using boxplots. Now consider the dataset we’ve been dealing with again and lets draw a boxplot on attribute preg

In [130]:

```
import matplotlib.pyplot as plt
y = list(data.preg)
plt.boxplot(y)
plt.show()
```



The output plot would look like this with spotting out outliers.

ANOVA using Python

What is ANOVA (ANalysis Of VAriance)?

•Used to compare the means of more than 2 groups (t-test can be used to compare 2 groups)
•Groups mean differences inferred by analyzing variances
•Main types: One-way (one factor) and two-way (two factors) ANOVA (factor is an independent variable)

ANOVA Hypotheses

•Null hypotheses: Groups means are equal (no variation in means of groups)
•Alternative hypotheses: At least, one group mean is different from other groups

ANOVA Assumptions

•Residuals (experimental error) are normally distributed
•Homogeneity of variances (variances are equal between treatment groups)
•Observations are sampled independently from each other

How ANOVA works?

•Check sample sizes: equal number of observation in each group(SS)
•Calculate Mean Square for each group (MS) (SS of group/level-1);
level-1 is a degree of freedom (df) for a group
•Calculate Mean Square error (MSE) (SS error/df of residuals)
•Calculate F-value (MS of group/MSE)

One-way (one factor) ANOVA

In [138]:
```
import pandas as pd
# load data file
d = pd.read_csv('Anova.csv')
```

In [139]:
```
d
```

Out[139]:

	A	B	C	D
0	25	45	30	54
1	30	55	29	60
2	28	29	33	51
3	36	56	37	62
4	29	40	27	73

Here, there are four treatments (A, B, C, and D), which are groups for ANOVA analysis. Treatments are independent variable and termed as factor. As there are four types of treatments, treatment factor has four levels.

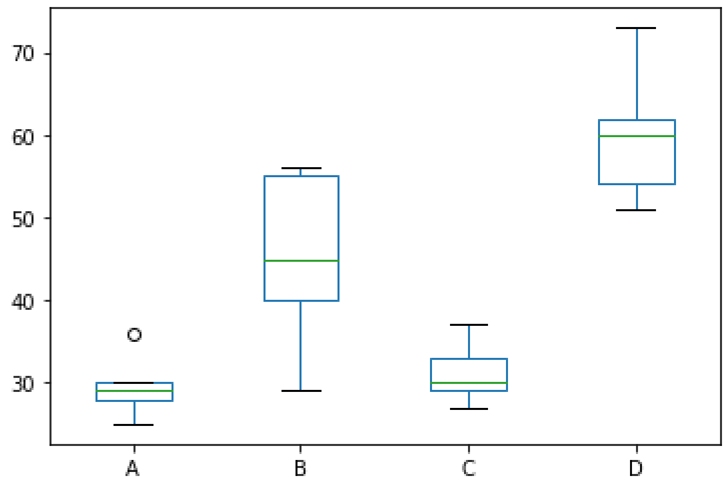
For this experimental design, there is only factor (treatments) or independent variable to evaluate, and therefore, one-way ANOVA is suitable for analysis.

In [140]:

```
# generate a boxplot to see the data distribution by treatments.
#Using boxplot, we can easily detect the differences between differ
d.boxplot(column=['A', 'B', 'C', 'D'], grid=False)
```

Out[140]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x26de189ab70>
```



In [144]:

```
# load packages
import scipy.stats as stats
# stats f_oneway functions takes the groups as input and returns F
fvalue, pvalue = stats.f_oneway(d['A'], d['B'], d['C'], d['D'])
print(fvalue, pvalue)
```

```
17.492810457516338 2.639241146210922e-05
```

Interpretation: The P-value obtained from ANOVA analysis is significant ($P<0.05$), and therefore, we conclude that there are significant differences among treatments.

Questionnaire

Why do we need missing value imputation over rows deletion?

The two major advantages of missing data imputation over rows deletion:
1.The variance of analyses based on imputed data is usually lower, since missing data imputation does not reduce your sample size.
2.Depending on the response mechanism, missing data imputation outperforms rows deletion in terms of bias.
Missing data imputation almost always improves the quality of our data.

Which value can be used for missing value imputation?

Often a simple, if not always satisfactory, choice for missing values that are known not to be zero is to use some 'central' value of the variable. This is often the mean, median, or mode, and thus usually has limited impact on the distribution. We might choose to use the mean, for example, if the variable is otherwise generally normally distributed (and in particular does not have any skewness). If the data does exhibit some skewness though (e.g., there are a small number of very large values) then the median might be a better choice.

For categoric variables, there is, of course, no mean nor median, and so in such cases we might choose to use the mode (the most frequent value) as the default to fill in for the otherwise missing values. The mode can also be used for numeric variables.