Tikrit University
Computer Science Dept.

Master Degree

Lecture -2-

Associate Professor

Dr. Eng. Zaidoon.T.AL-Qaysi

College of Computer Science and Mathematics

(2023-2024)

- **A data structure** is a specific way to organize and store data in a computer so that the data may be accessed and changed effectively. It is a grouping of data values of various types that preserves the logical dependencies among the data elements and the functions or operations that apply to the stored data for retrieval, access, and update.

- One of the most basic ways to think about data is whether it is **structured** or **not**. This is especially important for data science because most of the techniques that we will learn depend on one or the other inherent characteristic. Most commonly, **structured data** refers to highly organized information that can be seamlessly included in a database and readily searched via simple search operations; whereas **unstructured data** is essentially the opposite, devoid of any underlying structure.

- A **data attribute** defines a data element using a single descriptor with specific characteristics. In statistics, it is also called a measurement scale. The way of handling each attribute varies with its nature. Hence, it is a necessary as well as important to understand attribute types (Figure 1) before processing.

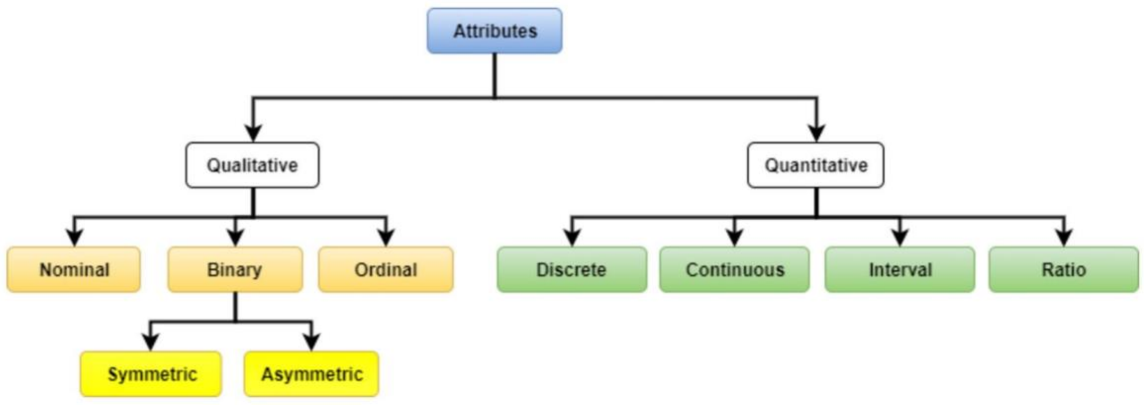


Figure 1: Attribute types in databases.

- **Qualitative Attribute:** As the name suggests, qualitative attributes are data types that cannot be measured using standard measuring units; rather, they can be observed, compared, and classified.

- **A nominal attribute** represents a specific symbol or the name of an entity; it is also called a categorical attribute. It is a qualitative attribute that cannot be ordered or ranked. No meaningful relationship can be established among the values. Only the logical or "is equal" operator can be applied among the values. Categorical values cannot be dealt with directly for various Data Science tasks. For example, neural networks are not built to handle categorical data straightforwardly. A few examples of nominal attributes are given in Table 1.

Table 1: Examples of nominal attributes and values

Attribute	**Values**
Color	Red, Blue, Green
City	Moscow, Kolkata, New York
Name	John, Harry, Ron

- A **binary attribute** holds discrete values with exactly two mutually exclusive states, either 1 (true) or 0 (false). A person is either infected by a virus or not infected. Both states never occur at the same time. A binary attribute is said to be a symmetric binary attribute if both states possess equal importance. In such a case, it hardly matters which attribute value, such as 0 or 1, to assign. On the other hand, an asymmetric binary attribute does not give equal importance to both states. Table 2 gives a few examples of binary attributes.

Table 2: Examples of binary attribute types

Attribute	Values	Type
Gender	Male, Female	Symmetric
Status	Present, Absent	Asymmetric

- **Ordinal Attribute** values that can be ordered or ranked are called ordinal attributes. It is worth mentioning that magnitude-wise, the difference between ordered values may not be known or may not even be significant. The ordering exhibits the importance of the attribute value but does not indicate how important. For example, the allowable values for an attribute may be Good, Better, and Best. One may rank the values based on their importance without being clear about how Good differs from Better or Best. A few more examples are listed in Table 3.

Table 3: Examples of ordinal attribute and values

Attribute	Values
Grade	A+, A, B+, B, C+, C
Position	Asst. Prof., Assoc. Prof., Prof.
Qualification	UG, PG, PhD

- Unlike Qualitative attributes, **Quantitative attributes** are measurable quantities. The values are quantified using straightforward numbers. Hence, they are also called numerical attributes. Statistical tools are easily applicable to such types of attributes for analysis. They can also be represented by data-visualization tools like pie charts, line and bar graphs, scatter plots, etc. The values may be countable or non-countable. Accordingly, they are of the two following types: Table 4 shows some quantitative attributes.

Table 4: Quantitative attributes

Attribute	Values	Type
Population	5000, 7680, 2500	Discrete
Price	3400.55, 6600.9	Continuous
Temperature	10°C–20°C, 20°C–30°C	Interval
Family Dependents	0, 2, 3	Ratio

- **Discrete Countable quantitative attributes** are also called discrete attributes. Usually, if the plural of the attribute name may be prefixed with “the number of,” it can be considered a discrete value attribute. Examples of discrete attributes include the number of persons, cars, buildings, population of a city, etc. They are denoted with whole numbers or integer values and can’t have fraction.

- **Continuous values** are non-countable and measurable quantities with infinite possible states. They are represented with fractional or floating point values. Attributes such as Weight, Height, Price, Blood Pressure, Temperature, etc., hold continuous values. Whether discrete or continuous type, quantitative attributes can be of the following two types.

- **Interval Attribute** Like ordinal attributes, quantitative attributes (both discrete and continuous) can be ordered. The distinction comes from whether differences or intervals among the ordered values are known or not. Interval values do not have any correct references or true zero points. The zero point is the value at which no quantity or amount of the attribute is measured. For example, temperature intervals (Table 4), such as 10◦C to 20◦C, represents a range of values within which temperature can vary. The temperatures for two consecutive days may differ by a certain degree, but we cannot say there is “no temperature” (zero-point).

- **Ratio attribute** is another quantitative attribute representing a relative or comparative quantity relating to two or more values. Unlike intervals, ratios have fixed zero points. A fixed zero point is a reference point on a fixed scale that does not vary based on the measured attribute. In other words, it is a point on the scale representing a constant value, regardless of the quantity or amount of the attribute being measured. Ratio-scale attributes can also be ordered, but with a perfect zero. However, no negative value is allowed in ratio-scale attributes. If a value is ratio scaled, it implies a multiple (or ratio) of another value. Parametric measures such as mean, median, mode, and quantile range are considered ratio-scale attributes. As given in Table 4, Family Dependents are considered Ratio data because they possess all the properties of interval data and, in addition, have a meaningful zero point.

- **Researchers and data scientists** frequently encounter circumstances where they either lack suitable real-world (primary or secondary) data or may be unable to use such data due to closed access, privacy issues, and/or litigation potential. The alternative solution in such cases is to use artificially generated datasets produced using synthetic-data generation.

- **Synthetic-data generation** is the process of creating artificial data as a replacement for realworld data, either manually or automatically, using computer simulations or algorithms. If the original data are unavailable, “fake” data may be created entirely from scratch or seeded with a real dataset.

- **Data augmentation** involves adding slightly altered copies of already-existing dataset samples. Using data augmentation, a dataset can be expanded with nearly identical samples but with slight differences in data characteristics. This is useful when adequate data samples are scarce. For example, adding new face images into a face database by altering the orientation, brightness, and shape of the faces augments the face database.

- **Data randomization** does not produce new data elements; it simply changes slightly the items already present in the dataset. With randomization, data attributes or features may be altered in some samples in the dataset. Randomization helps protect sensitive data by making it harder for unauthorized persons to identify or infer personal information from the data. By randomizing the order of the data, any patterns or relationships that could be used to identify individuals are disrupted. For example, a clinical dataset of cancer patients may be randomized by altering a few selected features or attributes of the real data in order to hide the patient's personal details.

- The intention behind synthetic-data generation is to provide alternatives to real data when real data are unavailable or have privacy, intellectual property, or other similar issues. Based on the amount of artificiality or syntheticity in the generated data, they can be classified into two types.

1. **Fully Synthetic:** Data that are fully synthetic and have no direct relationship to actual data. Even though all the necessary characteristics of real data are present, it is likely that there is no real individual or example that corresponds to a generated data example. When creating fully synthetic data, the data generator often finds the density function of the features in the real data and estimates the appropriate parameters for data generation. The bootstrap method and multiple approaches to imputation are common ways of producing fully synthetic data.

2. **Partially Synthetic:** All information from the real data is retained in partially synthetic data, with the exception of sensitive information. Only selected sensitive feature values are replaced with synthetic values in such data. Most real values are likely to be retained in the carefully curated synthetic-data collection because they are extracted from the real data. Multiple imputation methods and model-based procedures are used for producing partially synthetic data.

- The typical workflow of the data generation pipeline is shown in **Figure 2** , and their basic data generation step may follow five major steps:-

1. **Determine Objective:** The first step is to understand and establish the objectives for the planned synthetic-data generation and the techniques to be used later for data analysis. Further, one needs to understand organizational, legal, and other restrictions because data privacy is currently a major concern. For instance, if one needs clinical data for AIDS patients for analysis, confidentiality is a serious issue.

2. **Select Generator:** Next, one needs to select a data-generation model. For simulation, appropriate technical knowledge and adequate infrastructure are necessary. For example, generating artificial IoT traffic data using Monte Carlo simulation requires knowledge of various model hyperparameters to be tuned.

3. **Collect Sample Data:** Most synthetic data generators need real-data samples to learn the underlying probability distribution. The availability of carefully collected real-data samples determines the quality of the generated synthetic data.

4. **Train Model:** The selected data-generation model needs to be trained using collected real-data samples by tuning the hyperparameters. The target is to make the model understand well the sample data and create data close to real-data samples.

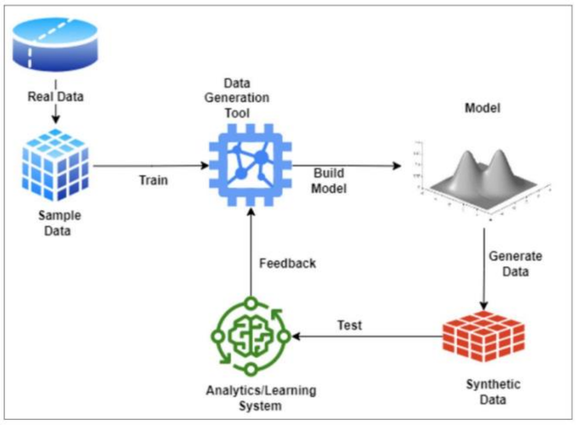


Figure 2: Synthetic-Data-Generation Workflow.

5. **Generate Data:** Once trained, synthetic datasets of any size can be generated using the trained model.
6. Evaluate the Data: Assessing the quality of generated synthetic data is important to ensure usability. The best way to evaluate this is to feed the data to data-analysis algorithms, learn a trained model, and test its quality with real-data samples. If the system performs according to expectation, the generated data can be used or archived. In the case of underperformance, the error may be fed back to the synthetic-data-generation model for further tuning.

- To **generate synthetic data**, the following techniques may be used:-

1. **Statistical Distribution:** The idea behind using a statistical distribution for synthetic data generation is first to learn the underlying probability distribution for real-data examples. The quality of synthetic data to be produced depends on sampling. If sampling is biased, naturally, the outcome may be ill-distributed and may not represent real data. There may be scenarios where no data samples are available. In such scenarios, expert judgment and experience matter greatly. A data scientist is able to generate random samples by following a probability distribution such as Gaussian, or Chi-square distribution.

2. **Agent Modeling:** First, an agent data generator is modeled to behave as if in real scenarios . Next, it uses a modeled agent for random fake-data generation. It may perform curve fitting to fit real data to a known distribution. Later, a suitable simulation method like Monte Carlo can be used for synthetic-data generation.

3. **Deep Neural Networks:** In recent years, deep neural-network-based techniques have become popular and effective for generating bulk and close-to-real data. Advanced machine-learning approaches, including deep models, can learn relevant features for a variety of data. Neural networks are especially well suited for creating synthetic data through trial and error methods. They can learn to duplicate the data and generalize to create a representation that may have characteristics of real data. Two popular deep neural architectures include Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN).

- **Data quality** refers to the state of the data in terms of completeness, correctness, and consistency. Assessing and ensuring data quality by removing errors and data inconsistencies can improve the input-data quality and makes it fit for the intended purpose.

- **Data** in the real world is often dirty; that is, it is in need of being cleaned up before it can be used for a desired purpose. This is often called data pre-processing.

- **Data preprocessing** comprises several essential steps, including **Data Cleaning**, **Data Reduction**, **Data Transformation, and Data Integration.** A schematic overview of the overall workflow of preprocessing steps is illustrated in Figure 3.

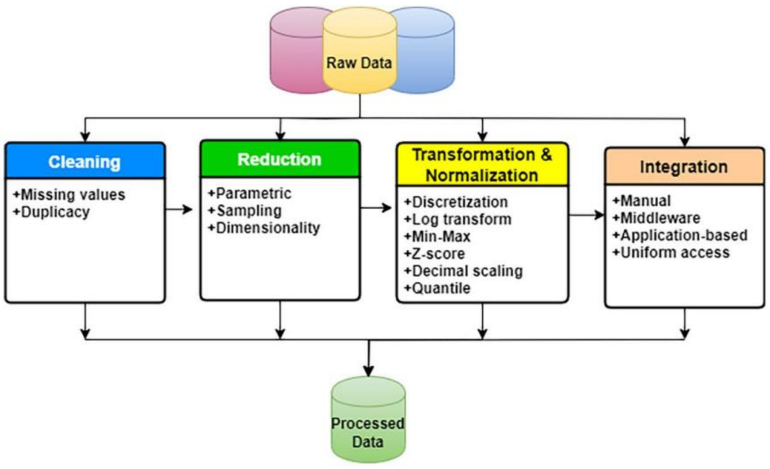


Figure 3: A schematic overview of the overall workflow of preprocessing steps

- Creating high-quality data involves a step called **data cleaning**. This step involves removing noise, estimating missing values, and correcting background information before normalizing the data. Data cleaning typically addresses two primary issues: missing values and data duplication. The latter issue can **increase computation time** without providing any additional value to the final result.

- Some of the factors that indicate that data is not clean or ready to process:

  1. **Incomplete**. When some of the attribute values are lacking, certain attributes of interest are lacking, or attributes contain only aggregate data.
  2. **Noisy**. When data contains errors or outliers. For example, some of the data points in a dataset may contain extreme values that can severely affect the dataset’s range.
  3. **Inconsistent**. Data contains discrepancies in codes or names. For example, if the “Name” column for registration records of employees contains values other than alphabetical letters, or if records do not start with a capital letter, discrepancies are present.

- Raw data must be pre-processed prior to being used to fit and evaluate a machine learning model. This step in a predictive modeling project is referred to as data preparation, although it goes by many other names, such as data wrangling, data cleaning, data pre-processing and feature engineering. Nevertheless, there are common or standard tasks that you may use or explore during the data preparation step in a machine learning project. These tasks include:

    1. **Data Cleaning:** Identifying and correcting mistakes or errors in the data. ˆ
    2. **Feature Selection:** Identifying those input variables that are most relevant to the task. ˆ
    3. **Data Transforms:** Changing the scale or distribution of variables. ˆ
    4. **Feature Engineering:** Deriving new variables from available data. ˆ
    5. **Dimensionality Reduction:** Creating compact projections of the data.

- **Data cleaning** is a critically important step in any machine learning project. In tabular data, there are many different statistical analysis and data visualization techniques you can use to explore your data in order to identify data cleaning operations you may want to perform. Before jumping to the sophisticated methods, there are some very basic data cleaning operations that you probably should perform on every single machine learning project. Basic data cleaning you should always perform on your dataset such as:

1. Identify columns that contain a single value.
2. Delete columns that contain a single value.
3. Consider columns that have very few values.
4. Remove columns that have a low variance.
5. Identify rows that contain duplicate data.
6. Delete rows that contain duplicate data

- Columns that have a single observation or value are probably useless for modeling. These columns or predictors are referred to zero-variance predictors as if we measured the variance (average value from the mean), it would be zero. Here, a single value means that each row for that column has the same value. For example, the column X1 has the value 1.0 for all rows in the dataset:

```
X1
1.0
1.0
1.0
1.0
1.0
...
```

Figure 4: Example of a column that contains a single value.

- Columns are relatively easy to remove from a NumPy array or Pandas DataFrame. One approach is to record all columns that have a single unique value, then delete them from the Pandas DataFrame by calling the drop() function. The complete example is listed below.

```
# delete columns with a single unique value
from pandas import read_csv
# load the dataset
df = read_csv('oil-spill.csv', header=None)
print(df.shape)
# get number of unique values for each column
counts = df.nunique()
# record columns to delete
to_del = [i for i,v in enumerate(counts) if v == 1]
print(to_del)
# drop useless columns
df.drop(to_del, axis=1, inplace=True)
print(df.shape)
```

```
(937, 50)
[22]
(937, 49)
```

Figure 5: Example of deleting columns that have a single value.

Running the example in **Figure 6**, we can see that 11 of the 50 variables have numerical variables that have unique values that are less than 1 percent of the number of rows. This does not mean that these rows and columns should be deleted, but they require further attention.

```
# summarize the percentage of unique values for each column using numpy
from numpy import loadtxt
from numpy import unique
# load the dataset
data = loadtxt('oil-spill.csv', delimiter=',')
# summarize the number of unique values in each column
for i in range(data.shape[1]):
  num = len(unique(data[:, i]))
  percentage = float(num) / data.shape[0] * 100
  if percentage < 1:
    print('%d, %d, %.1f%%' % (i, num, percentage))
```

Figure 6: Example of reporting on columns with low variance.,

```
21, 9, 1.0%
22, 1, 0.1%
24, 9, 1.0%
25, 8, 0.9%
26, 9, 1.0%
32, 4, 0.4%
36, 3, 0.3%
38, 9, 1.0%
39, 9, 1.0%
45, 2, 0.2%
49, 2, 0.2%
```

Figure 7: output of code in figure 6 reporting on columns with low variance.

For example, if we wanted to delete all 11 columns with unique values less than 1 percent of rows; the example below demonstrates this.

```
# delete columns where number of unique values is less than 1% of the rows
from pandas import read_csv
# load the dataset
df = read_csv('oil-spill.csv', header=None)
print(df.shape)
# get number of unique values for each column
counts = df.nunique()
# record columns to delete
to_del = [i for i,v in enumerate(counts) if (float(v)/df.shape[0]*100) < 1]
print(to_del)
# drop useless columns
df.drop(to_del, axis=1, inplace=True)
print(df.shape)
```

Figure 8: Example of removing columns with low variance

Running the example in **Figure 8** first loads the dataset and reports the number of rows and columns. The number of unique values for each column is calculated, and those columns that have a number of unique values less than 1 percent of the rows are identified. In this case, 11 columns. The identified columns are then removed from the DataFrame, and the number of rows and columns in the DataFrame are reported to confirm the change.

```
(937, 50)
[21, 22, 24, 25, 26, 32, 36, 38, 39, 45, 49]
(937, 39)
```

Figure 9: Output of figure 8 for removing columns with low variance.

Another approach to the problem of removing columns with few unique values is to consider the variance of the column. Recall that the variance is a statistic calculated on a variable as the average squared difference of values in the sample from the mean. The variance can be used as a filter for identifying columns to be removed from the dataset. A column that has a single value has a variance of 0.0, and a column that has very few unique values may have a small variance. The VarianceThreshold class from the scikit-learn library supports this as a type of feature selection. An instance of the class can be created and we can specify the threshold argument, which defaults to 0.0 to remove columns with a single value. It can then be fit and applied to a dataset by calling the fit transform() function to create a transformed version of the dataset where the columns that have a variance lower than the threshold have been removed automatically.

```
# example of applying the variance threshold for feature selection
from pandas import read_csv
from sklearn.feature_selection import VarianceThreshold
# load the dataset
df = read_csv('oil-spill.csv', header=None)
# split data into inputs and outputs
data = df.values
X = data[:, :-1]
y = data[:, -1]
print(X.shape, y.shape)
# define the transform
transform = VarianceThreshold()
# transform the input data
X_sel = transform.fit_transform(X)
print(X_sel.shape)
```

Figure 10: Example of removing columns that have a low variance.

Running the example first loads the dataset, then applies the transform to remove all columns with a variance of 0.0. The shape of the dataset is reported before and after the transform, and we can see that the single column where all values are the same has been removed.

```
(937, 49) (937,)
(937, 48)
```

Figure 11: Example output from removing columns that have a low variance

We can expand this example and see what happens when we use different thresholds. We can define a sequence of thresholds from 0.0 to 0.5 with a step size of 0.05, e.g. 0.0, 0.05, 0.1, etc.

```python
# explore the effect of the variance thresholds on the number of selected features
from numpy import arange
from pandas import read_csv
from sklearn.feature_selection import VarianceThreshold
from matplotlib import pyplot
# load the dataset
df = read_csv('oil-spill.csv', header=None)
# split data into inputs and outputs
data = df.values
X = data[:, :-1]
y = data[:, -1]
print(X.shape, y.shape)
# define thresholds to check
thresholds = arange(0.0, 0.55, 0.05)
# apply transform with each threshold
results = list()
for t in thresholds:
  # define the transform
  transform = VarianceThreshold(threshold=t)
  # transform the input data
  X_sel = transform.fit_transform(X)
  # determine the number of input features
  n_features = X_sel.shape[1]
  print('>Threshold=%.2f, Features=%d' % (t, n_features))
  # store the result
  results.append(n_features)
# plot the threshold vs the number of selected features
pyplot.plot(thresholds, results)
pyplot.show()
```

Figure 12: The effect of different variance thresholds on the number of features in the transformed dataset.

```
(937, 49) (937,)
>Threshold=0.00, Features=48
>Threshold=0.05, Features=37
>Threshold=0.10, Features=36
>Threshold=0.15, Features=35
>Threshold=0.20, Features=35
>Threshold=0.25, Features=35
>Threshold=0.30, Features=35
>Threshold=0.35, Features=35
>Threshold=0.40, Features=35
>Threshold=0.45, Features=33
>Threshold=0.50, Features=31
```

Figure 13: Output code in figure 12 for reviewing the effect of different variance thresholds

A line plot is then created showing the relationship between the threshold and the number of features in the transformed dataset. We can see that even with a small threshold between 0.15 and 0.4, that a large number of features (14) are removed immediately.

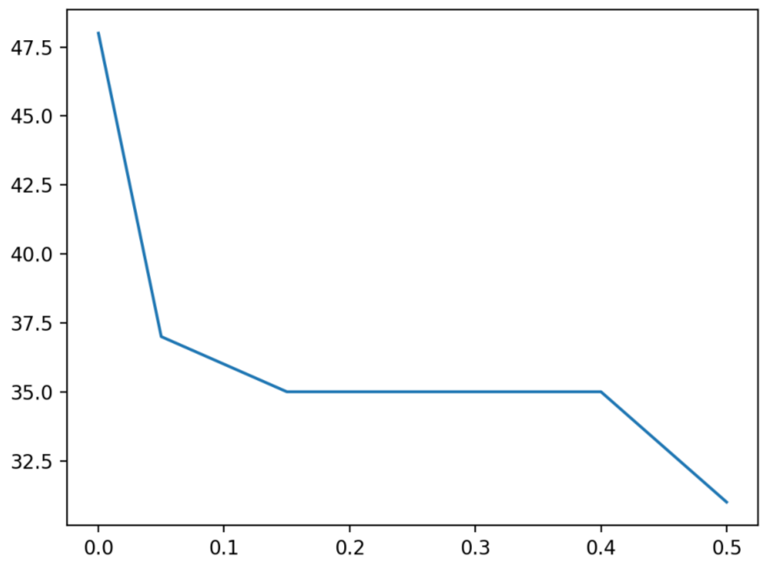


Figure 14: Variance Threshold versus Number of Selected Features.

- Rows of duplicate data should probably be deleted from your dataset prior to modeling. There are many ways to achieve this, although Pandas provides the drop duplicates() function that achieves exactly this. The example below demonstrates deleting duplicate rows from a dataset.

```
# delete rows of duplicate data from the dataset
from pandas import read_csv
# load the dataset
df = read_csv('iris.csv', header=None)
print(df.shape)
# delete duplicate rows
df.drop_duplicates(inplace=True)
print(df.shape)
```

```
(150, 5)
(147, 5)
```

Figure 15: Example of removing duplicate rows.

- Sometimes a dataset can contain extreme values that are outside the range of what is expected and unlike the other data. These are called **outliers** and often machine learning modeling and model skill in general can be improved by understanding and even removing these outlier values.

- An **outlier** is an observation that is unlike the other observations. They are rare, distinct, or do not fit in some way. Outliers can have many causes, such as: ˆ

  1. Measurement or input error. ˆ
  2. Data corruption. ˆ
  3. True outlier observation.

- A distribution of data refers to the shape it has when you graph it, such as with a histogram. The most commonly seen and therefore well-known distribution of continuous values is the **bell curve**. It is known as the **normal distribution**, because it the distribution that a lot of data falls into. Some examples of observations that have a Gaussian distribution include: ˆ
  1. People's heights. ˆ
  2. IQ scores. ˆ
  3. Body temperature.

- To explore some important summary statistics for data with a Gaussian distribution, two key parameters are used to define any Gaussian distribution; they are the **mean** and the **standard deviation**.

- The **central tendency** of a distribution refers to the middle or typical value in the distribution. The most common or most likely value. In the Gaussian distribution, the central tendency is called the **mean**, or more formally, the arithmetic mean, and is one of the two main parameters that define any Gaussian distribution. The mean of a sample is calculated as the sum of the observations divided by the total number of observations in the sample.

$$mean(x) = \frac{\sum_{i=1}^{n} x_i}{n}$$

Where x_i is the i^{th} observation from the dataset and n is the total number of observations.

It is also written in a more compact form as:

$$mean(x) = \frac{1}{n} \times \sum_{i=1}^{n} x_i$$

```
# calculate the mean of a sample
from numpy.random import seed
from numpy.random import randn
from numpy import mean
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate mean
result = mean(data)
print('Mean: %.3f' % result)
```

```
Mean: 50.049
```

Figure 16: Example of calculating the mean of a data sample

- The **mean** is easily influenced by outlier values, that is, rare values far from the mean.

- In the case of outliers or a non-Gaussian distribution, an alternate and commonly used central tendency to calculate is the **median**. The median is calculated by first sorting all data and then locating the middle value in the sample. This is straightforward if there are an odd number of observations. If there is an even number of observations, the median is calculated as the average of the middle two observations.

```
# calculate the median of a sample
from numpy.random import seed
from numpy.random import randn
from numpy import median
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate median
result = median(data)
print('Median: %.3f' % result)
```

```
Median: 50.042
```

Figure 17: Example of calculating the median of a data sample.

- The variance of a distribution refers to how much on average that observations vary or differ from the mean value. It is useful to think of the variance as a measure of the spread of a distribution. A low variance will have values grouped around the mean (e.g. a narrow bell shape), whereas a high variance will have values spread out from the mean (e.g. a wide bell shape.).

  The variance of a data sample drawn from a Gaussian distribution is calculated as the average squared difference of each observation in the sample from the sample mean:

$$variance(x) = \frac{1}{n-1} \times \sum_{i=1}^{n} (x_i - mean(x))^2$$

```
# generate and plot gaussians with different variance
from numpy import arange
from matplotlib import pyplot
from scipy.stats import norm
# x-axis for the plot
x_axis = arange(-3, 3, 0.001)
# plot low variance
pyplot.plot(x_axis, norm.pdf(x_axis, 0, 0.5))
# plot high variance
pyplot.plot(x_axis, norm.pdf(x_axis, 0, 1))
pyplot.show()
```

Figure 18: Gaussian numbers with different variances.

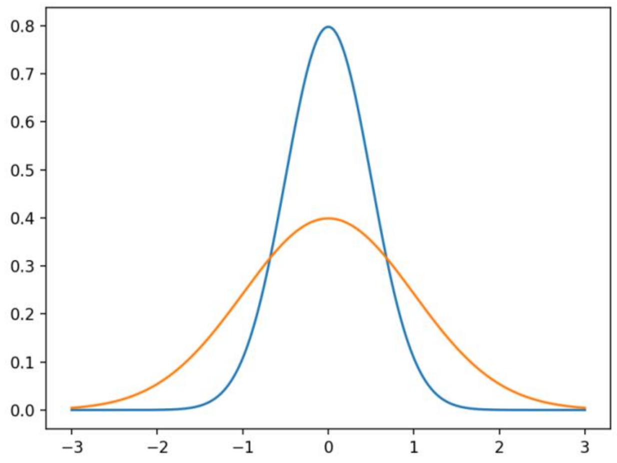


Figure 19: Line plots of Gaussian distributions with different variances.

```python
# calculate the variance of a sample
from numpy.random import seed
from numpy.random import randn
from numpy import var
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate variance
result = var(data)
print('Variance: %.3f' % result)
```

Figure 20: Calculating the variance of a data sample.

- Often, when the spread of a Gaussian distribution is summarized, it is described using the square root of the variance. This is called the **standard deviation**. We can wrap the variance calculation in a square root to calculate the standard deviation directly.

$$stdev(x) = \sqrt{\frac{1}{n-1} \times \sum_{i=1}^{n}(x_i - mean(x))^2}$$

Or:

$$stdev(x) = \sqrt{variance(x)}$$

```python
# calculate the standard deviation of a sample
from numpy.random import seed
from numpy.random import randn
from numpy import std
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate standard deviation
result = std(data)
print('Standard Deviation: %.3f' % result)
```

```
Standard Deviation: 4.994
```

Figure 21: Example of calculating the standard deviation of a data sample.

- If we know that the distribution of values in the sample is Gaussian or Gaussian-like, we can use the standard deviation of the sample as a cut-off for identifying outliers. The Gaussian distribution has the property that the standard deviation from the mean can be used to reliably summarize the percentage of values in the sample. For example, within one standard deviation of the mean will cover 68 percent of the data. So, if the mean is 50 and the standard deviation is 5, as in the test dataset above, then all data in the sample between 45 and 55 will account for about 68 percent of the data sample. We can cover more of the data sample if we expand the range as follows: ˆ

1. Standard Deviation from the Mean: 68 percent. ˆ
2. Standard Deviations from the Mean: 95 percent. ˆ
3. Standard Deviations from the Mean: 99.7 percent.

- A value that falls outside of 3 standard deviations is part of the distribution, but it is an unlikely or rare event at approximately 1 in 370 samples. Three standard deviations from the mean is a common cut-off in practice for identifying outliers in a Gaussian or Gaussian-like distribution. For smaller samples of data, perhaps a value of 2 standard deviations (95 percent) can be used, and for larger samples, perhaps a value of 4 standard deviations (99.9 percent) can be used.

```
# identify outliers with standard deviation
from numpy.random import seed
from numpy.random import randn
from numpy import mean
from numpy import std
# seed the random number generator
seed(1)
# generate univariate observations
data = 5 * randn(10000) + 50
# calculate summary statistics
data_mean, data_std = mean(data), std(data)
# define outliers
cut_off = data_std * 3
lower, upper = data_mean - cut_off, data_mean + cut_off
# identify outliers
outliers = [x for x in data if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))
# remove outliers
outliers_removed = [x for x in data if x >= lower and x <= upper]
print('Non-outlier observations: %d' % len(outliers_removed))
```

```
Identified outliers: 29
Non-outlier observations: 9971
```

Figure 22: Identifying and removing outliers using the standard deviation.

- A one-class classifier aims at capturing characteristics of training instances, in order to be able to distinguish between them and potential outliers to appear. A simple approach to identifying outliers is to locate those examples that are far from the other examples in the multi-dimensional feature space. This can work well for feature spaces with low dimensionality (few features), although it can become less reliable as the number of features is increased, referred to as the curse of dimensionality. The local outlier factor, or LOF for short, is a technique that attempts to harness the idea of nearest neighbors for outlier detection. Each example is assigned a scoring of how isolated or how likely it is to be outliers based on the size of its local neighborhood. Those examples with the largest score are more likely to be outliers. The scikit-learn library provides an implementation of this approach in the LocalOutlierFactor class.

```
# load and summarize the dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
# load the dataset
df = read_csv('housing.csv', header=None)
# retrieve the array
data = df.values
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# summarize the shape of the dataset
print(X.shape, y.shape)
# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# summarize the shape of the train and test sets
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(506, 13) (506,)
(339, 13) (167, 13) (339,) (167,)
```

Figure 23: Example of loading and summarizing the regression dataset.

```
# evaluate model on the raw dataset
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
# load the dataset
df = read_csv('housing.csv', header=None)
# retrieve the array
data = df.values
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# fit the model
model = LinearRegression()
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

```
MAE: 3.417
```

Figure 24: Example of evaluating a model on the regression dataset.

- Next, we can try removing outliers from the training dataset. The expectation is that the outliers are causing the linear regression model to learn a bias or skewed understanding of the problem, and that removing these outliers from the training set will allow a more effective model to be learned. We can achieve this by defining the LocalOutlierFactor model and using it to make a prediction on the training dataset, marking each row in the training dataset as normal (1) or an outlier (-1). We will use the default hyperparameters for the outlier detection model.

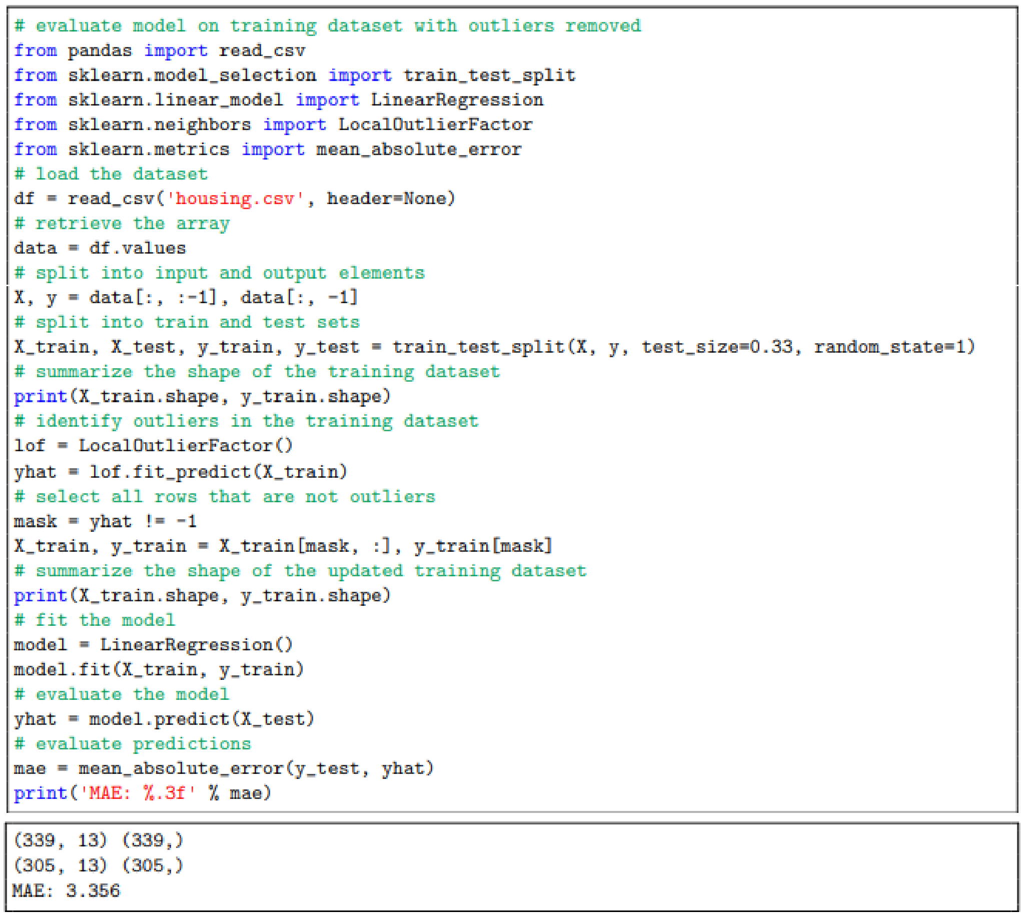
```
...
# identify outliers in the training dataset
lof = LocalOutlierFactor()
yhat = lof.fit_predict(X_train)
```

```
...
# select all rows that are not outliers
mask = yhat != -1
X_train, y_train = X_train[mask, :], y_train[mask]
```

Figure 25: Example of removing identified outliers from the dataset.

- We can then fit and evaluate the model as per normal. The updated example of evaluating a linear regression model with outliers deleted from the training dataset is listed below. Running the example fits and evaluates the linear regression model with outliers deleted from the training dataset.

- **Note:** Your specific results may vary given the stochastic nature of the learning algorithm, the evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average performance. Firstly, we can see that the number of examples in the training dataset has been reduced from 339 to 305, meaning 34 rows containing outliers were identified and deleted. We can also see a reduction in MAE from about 3.417 by a model fit on the entire training dataset, to about 3.356 on a model fit on the dataset with outliers removed.



```python
# evaluate model on training dataset with outliers removed
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import LocalOutlierFactor
from sklearn.metrics import mean_absolute_error
# load the dataset
df = read_csv('housing.csv', header=None)
# retrieve the array
data = df.values
# split into input and output elements
X, y = data[:, :-1], data[:, -1]
# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)
# summarize the shape of the training dataset
print(X_train.shape, y_train.shape)
# identify outliers in the training dataset
lof = LocalOutlierFactor()
yhat = lof.fit_predict(X_train)
# select all rows that are not outliers
mask = yhat != -1
X_train, y_train = X_train[mask, :], y_train[mask]
# summarize the shape of the updated training dataset
print(X_train.shape, y_train.shape)
# fit the model
model = LinearRegression()
model.fit(X_train, y_train)
# evaluate the model
yhat = model.predict(X_test)
# evaluate predictions
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

```
(339, 13) (339,)
(305, 13) (305,)
MAE: 3.356
```

Figure 26: Evaluating a model on the regression dataset with outliers removed