Python Intermediate

20th & 21st November 2021

by

Mogana Darshini Ganggayah, *PhD*

TIOBE Index for July 2021

July Headline: C, Java, and Python compete for the first position

The TIOBE index is celebrating its 20th anniversary this month. Back in 2001, the first TIOBE index was published as a personal hobby project to see what languages were in demand. The top 3 of the first TIOBE index were Java, C, and C++. Today's story looks strikingly similar. The first 3 programming languages are C, Java, and Python now. It is interesting to see that these 3 languages are getting closer than ever before. The difference between position 1 and position 3 is only 0.67%. This means that the next few months will be exciting. What language is going to win this battle? Python seems to have the best chances to become number 1, thanks to its market leadership in the booming field of data mining and artificial intelligence. Other interesting moves this month are: C++ is gaining more than 0.5% and is getting closer to the top 3, the Go language goes from position #20 to position #13, Rust from #30 to #27, TypeScript from #45 to #37, and Haskell from #49 to #39. -- *Paul Jansen CEO TIOBE Software*

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

Jul 2021	Jul 2020	Change	Programming Language	Ratings	Change
1	1		C	11.62%	-4.83%
2	2		Java	11.17%	-3.93%
3	3		Python	10.95%	+1.86%

TIOBE Index for November 2021

Nov 2021	Nov 2020	Change	Programming Language	Ratings	Change
1	2	↑	Python	11.77%	-0.35%
2	1	↓	C	10.72%	-5.49%
3	3		Java	10.72%	-0.96%
4	4		C++	8.28%	+0.69%
5	5		C#	6.06%	+1.39%
6	6		Visual Basic	5.72%	+1.72%
7	7		JavaScript	2.66%	+0.63%
8	16	⇈	Assembly language	2.52%	+1.35%
9	10	↑	SQL	2.11%	+0.58%
10	8	↓	PHP	1.81%	+0.02%

COURSE CONTENT

DAY 1

- Introduction to data mining
- Data mining essentials
- Knowledge discovery in database (KDD)
- Exploratory data analysis (EDA)
- Data cleaning
- Creating data frames
- Determine important features
- Feature to feature relationship
- Quantitative and qualitative data relationship

DAY 2

- Introduction to deep learning
- Data, model, objective function, optimization algorithm
- Linear model
- Activation function
- Data preparation (input and output)
- Weights specification
- ANN class and model training

DATA MINING

Data Deluge!

- Data is everywhere and it is expanding exponentially.
- Data is being generated from **various sources** in multiple formats.
- There is a significant demand for **skill** that knows how to manage, process, **analyze**, predict and discover knowledge from data to:
  - ✓ Solve problems
  - ✓ Enhance existing systems
  - ✓ Discover new technology

DATA MINING

	A	B	C	D	E
1		***Subject***			
2	***Name***	**Math**	**Physics**	**Chemistry**	**Biology**
3	Matt	38	58	66	49
4	Bob	88	92	74	90
5	Tom	57	77	91	91
6	Brad	82	56	45	95
7	Jenny	55	55	65	75
8	Maria	44	69	80	90
9	Jill	75	51	57	84
10	Josh	38	37	51	56

What I need to know from this data?

How to summarise the score?

What is the average score?

DATA MINING ESSENTIALS

New

Should generate new knowledge, patterns, relationship from data

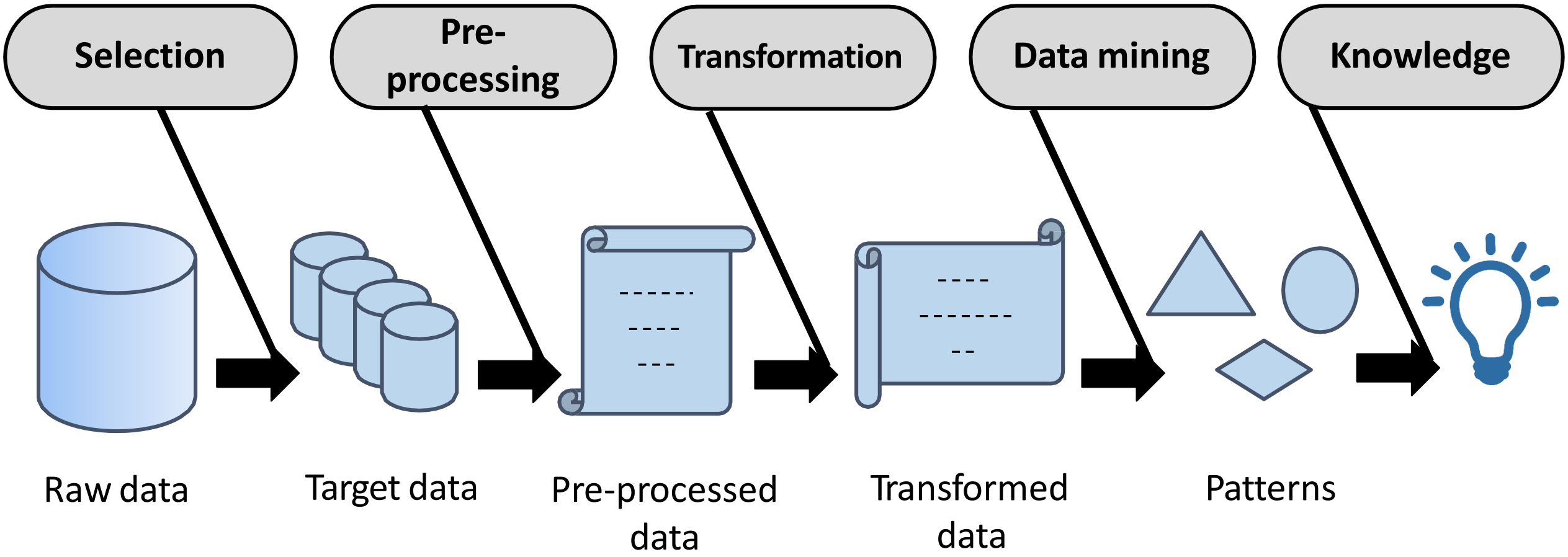
Correct

All the mined information might not be correct. Hence, validation need to be done in order to make sure the accuracy of data is good enough

Useful

The mined information from raw data should be practical, logic, useful and relevant to the users

KNOWLEDGE DISCOVERY IN DATABASE (KDD)



KNOWLEDGE DISCOVERY IN DATABASE (KDD)

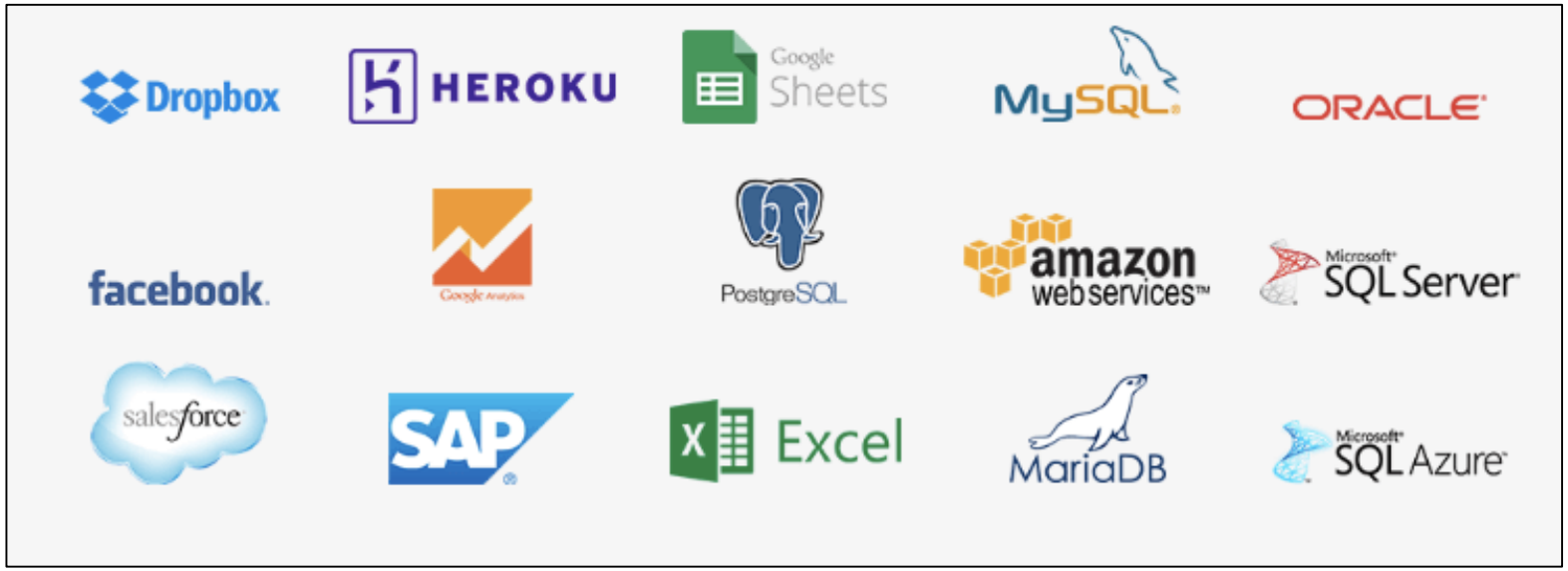
Selection

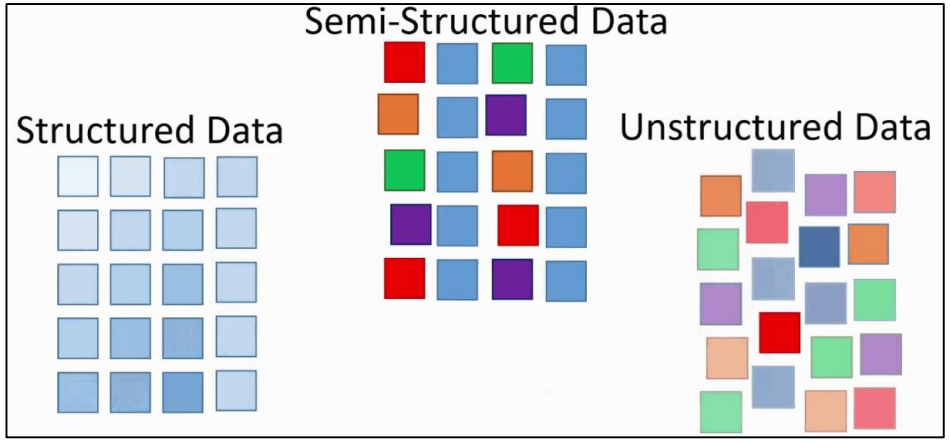
Pre-processing

Transformation

Data mining

Knowledge





Select data from multiple sources, ensure the type of data for further analysis

KNOWLEDGE DISCOVERY IN DATABASE (KDD)

Selection

Pre-processing

Transformation

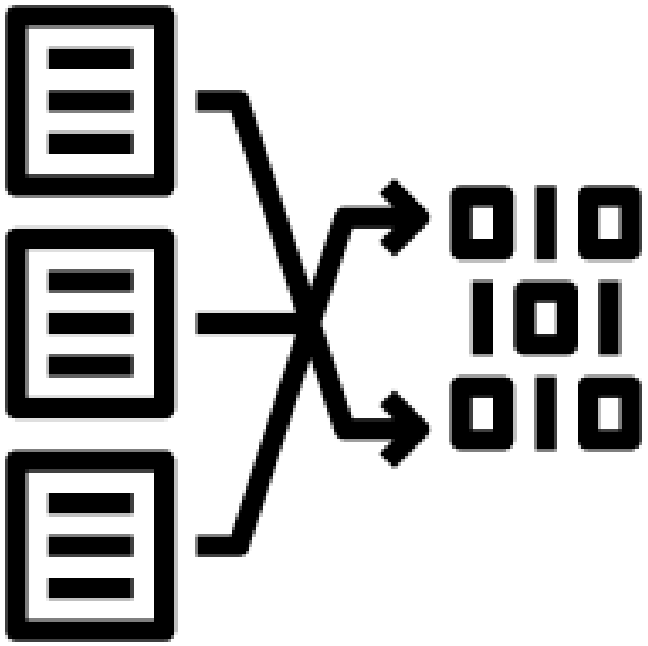
Data mining

Knowledge

- The selected data must be appropriate for mining task.
- Operations such as summarising and normalizing(cleansing) can be done to transform the data, such that it is suitable for mining.

KNOWLEDGE DISCOVERY IN DATABASE (KDD)

- Selection
- Pre-processing
- **Transformation**
- Data mining
- Knowledge



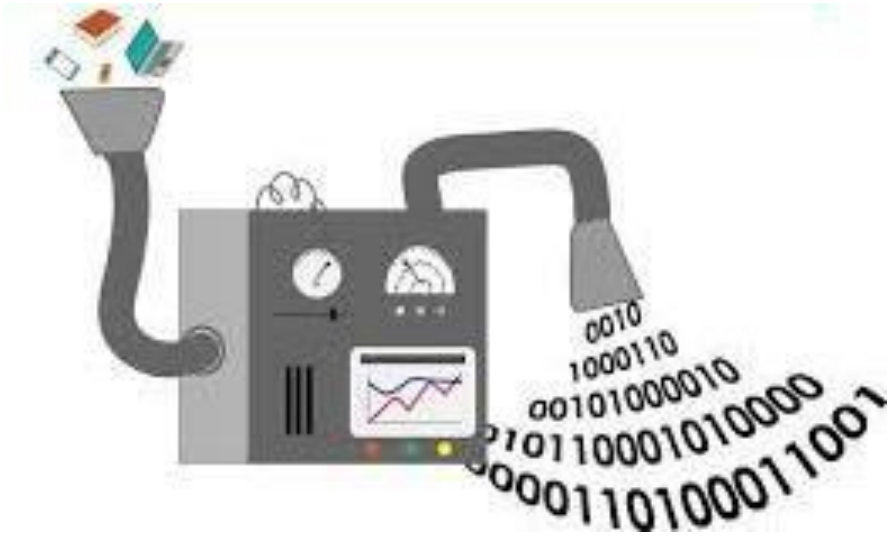
Transform the pre-processed data based on requirements, select:

- Tables
- Rows
- Columns
- Variables/ features/ parameters

that need to be analysed

KNOWLEDGE DISCOVERY IN DATABASE (KDD)

- Selection
- Pre-processing
- Transformation
- **Data mining**
- Knowledge



Apply data mining techniques such as clustering, classification, regression and prediction in order to extract patterns/ knowledge from the pre-processed data

KNOWLEDGE DISCOVERY IN DATABASE (KDD)

Selection

Pre-processing

Transformation

Data mining

Knowledge

The extracted patterns or information must be represented through visualisations such as graphs/tables/diagrams to make the users understand the output

EXPLORATORY DATA ANALYSIS

Education

Government

Media & Entertainment

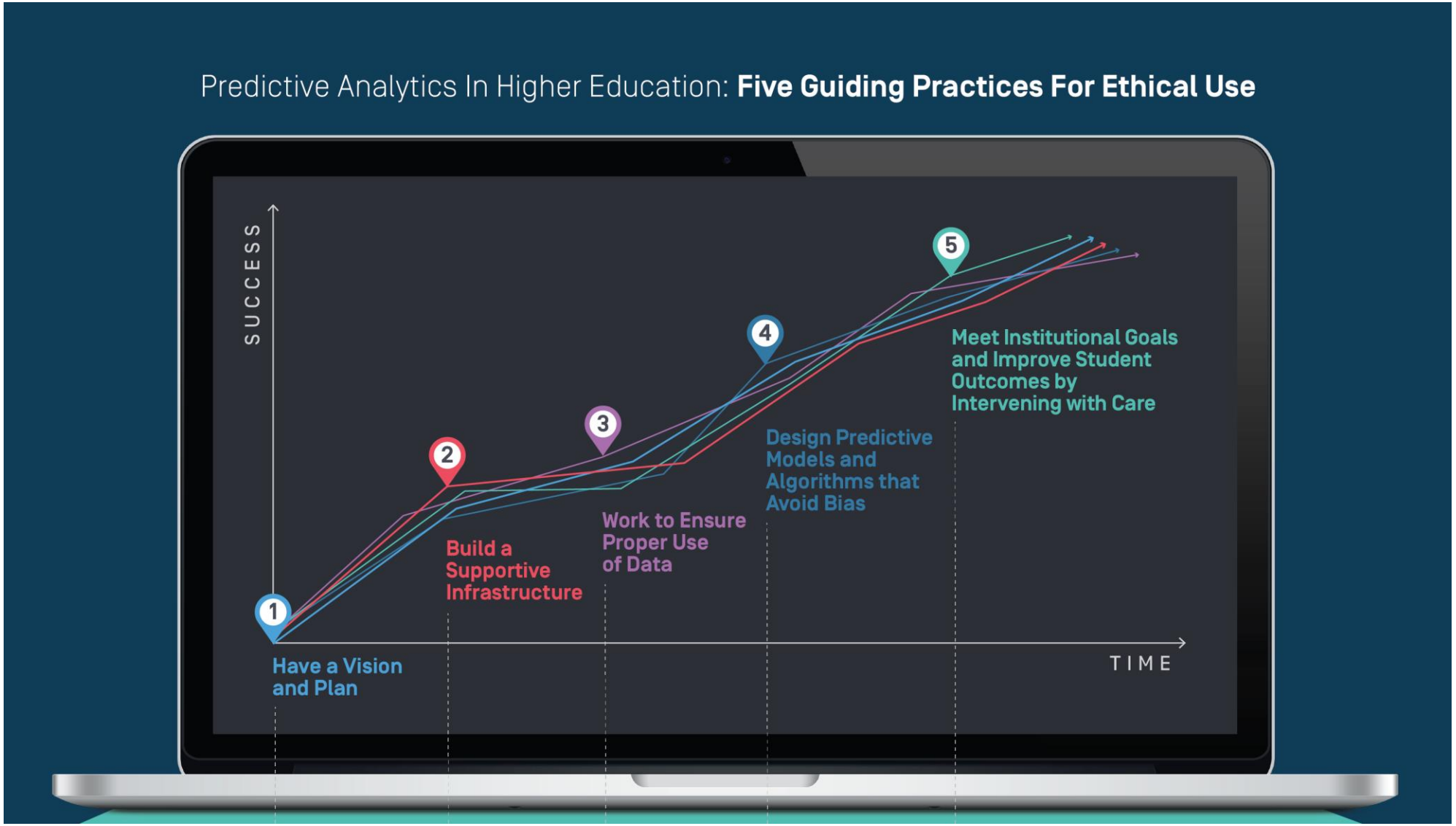
Healthcare

E-commerce

IOT

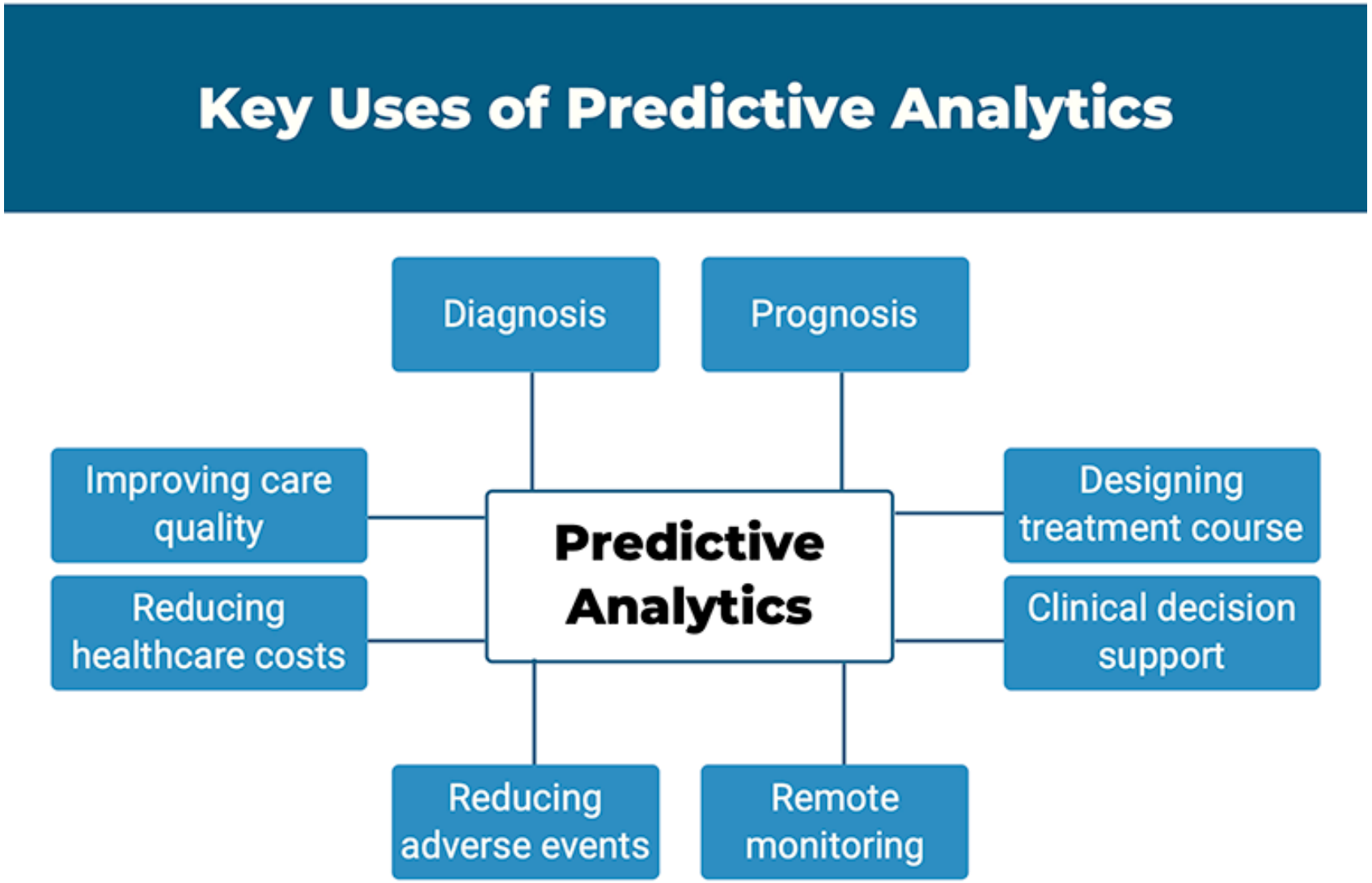
EXPLORATORY DATA ANALYSIS

Education



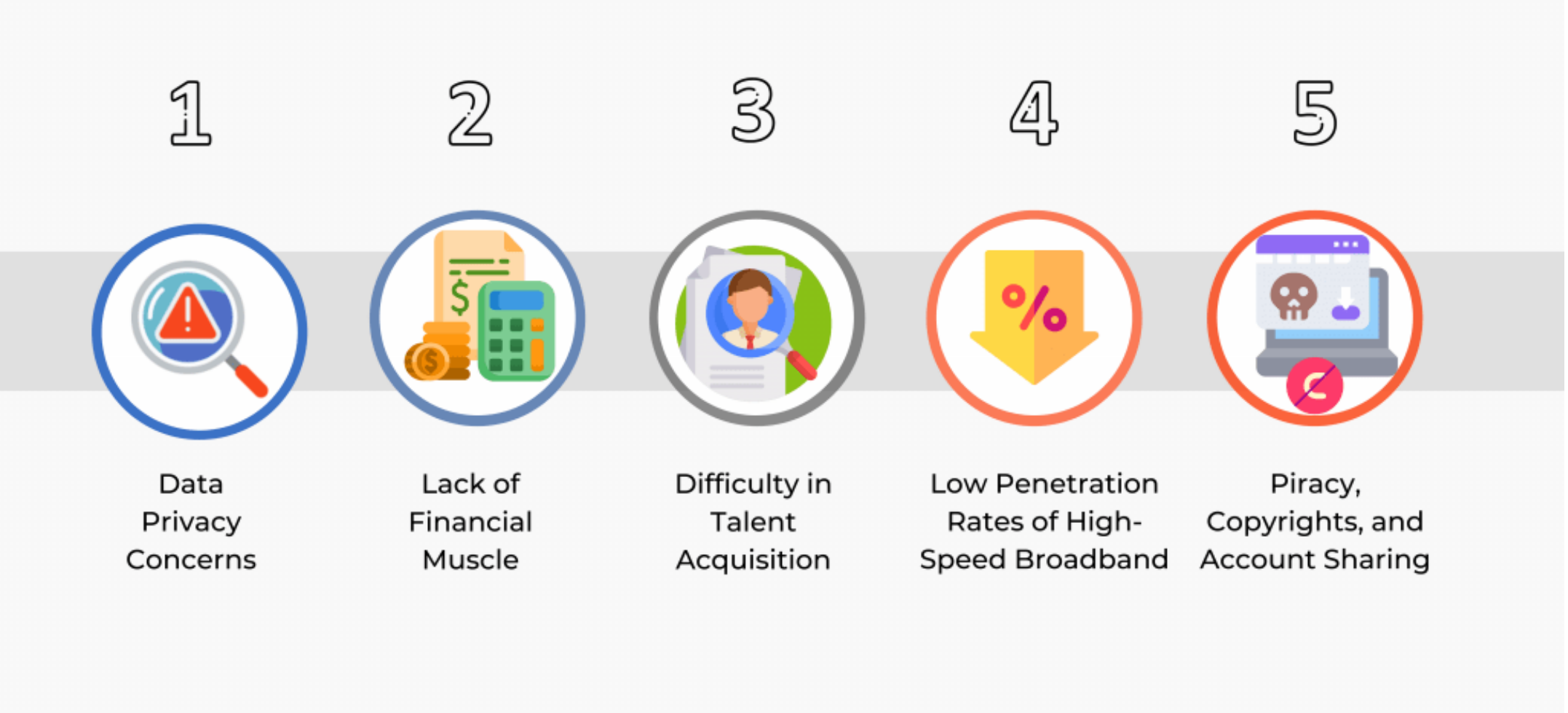
EXPLORATORY DATA ANALYSIS

Healthcare



EXPLORATORY DATA ANALYSIS

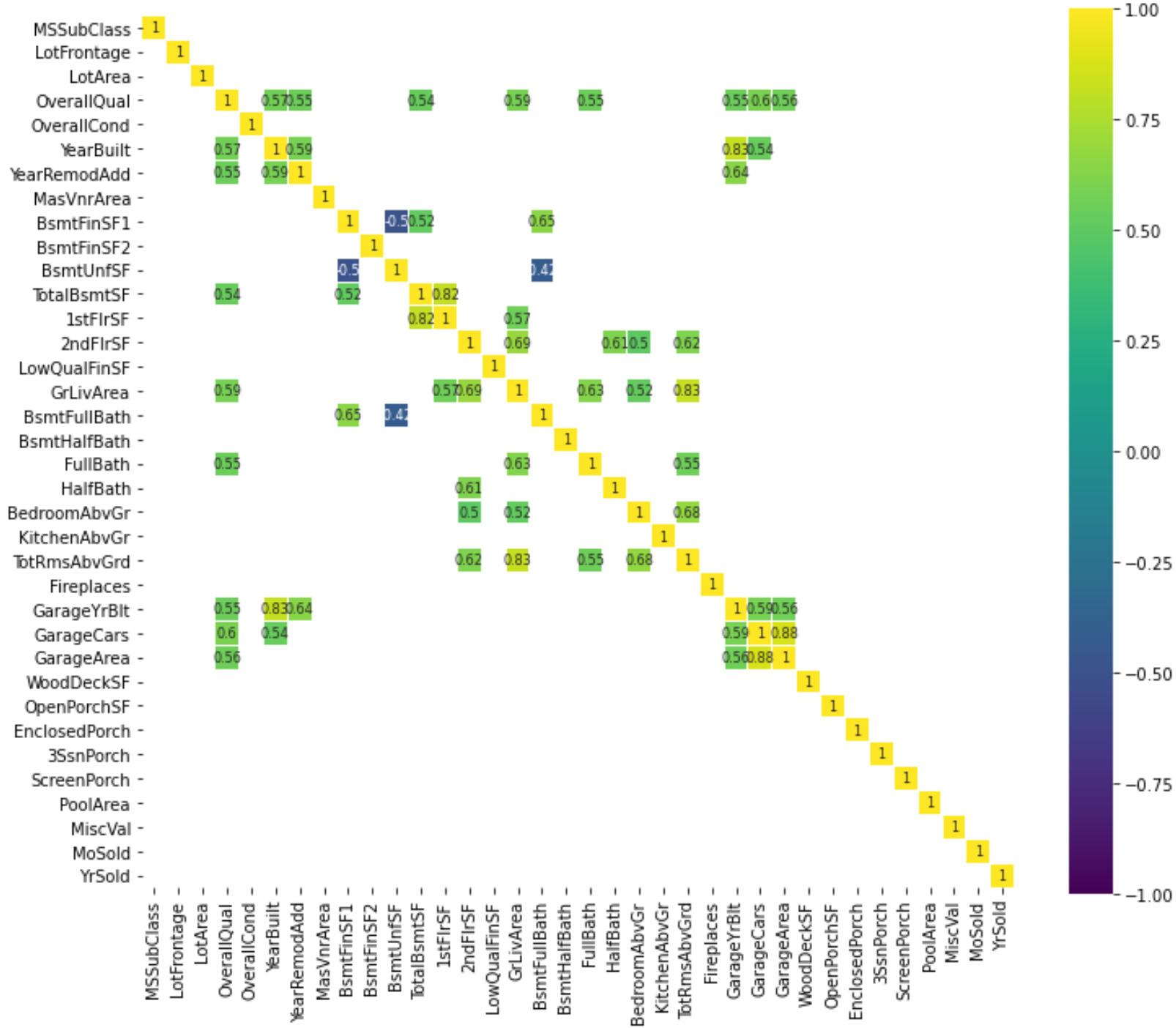
Media & entertainment



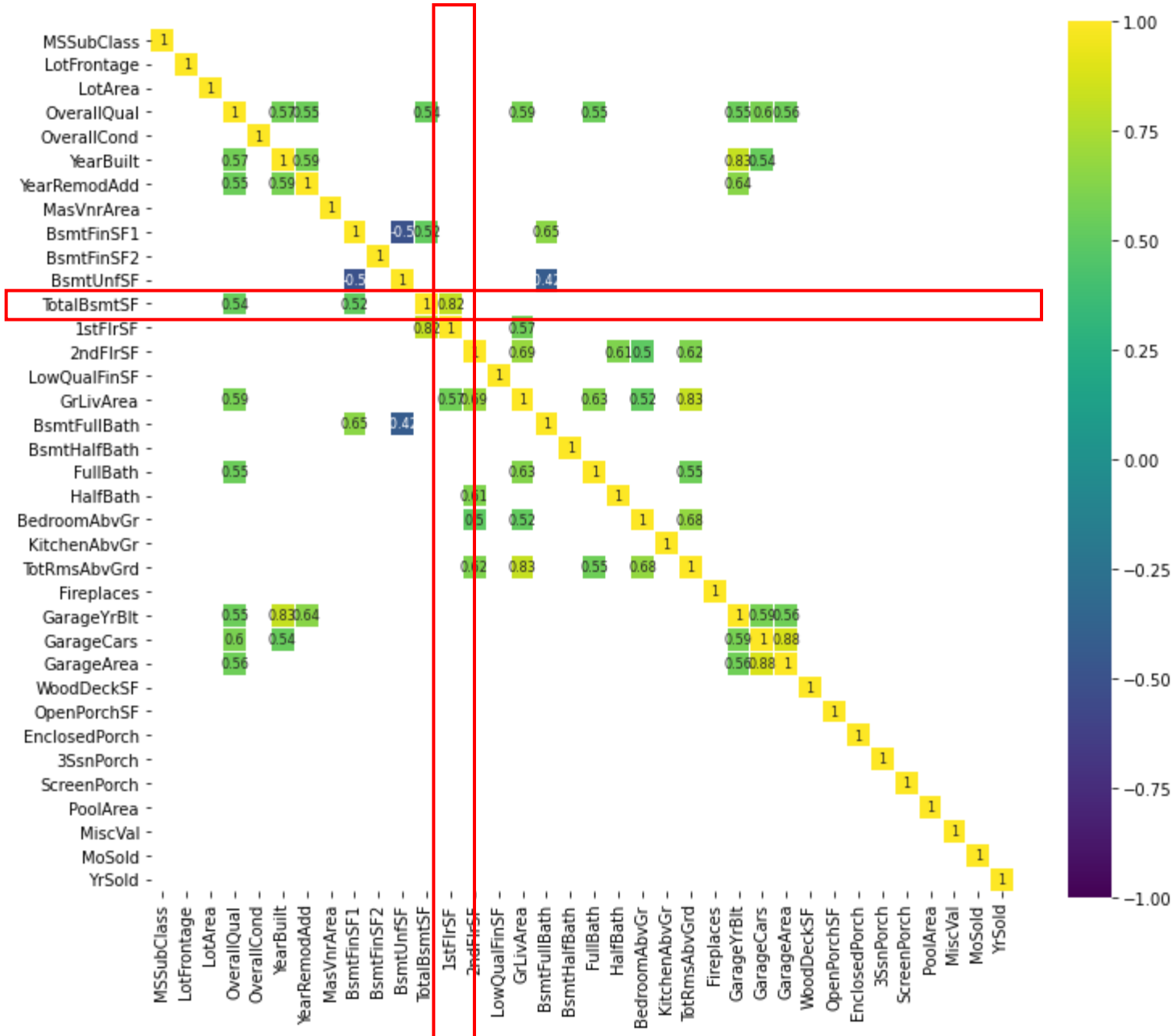
EXPLORATORY DATA ANALYSIS

EXPLORATORY DATA ANALYSIS

1. Describe the data
2. Clean the data – remove features based on missing value percentage
3. Create a data frame for numerical features
4. Determine strongly correlated features
5. Visualize feature to feature relationship
6. Quantitative and qualitative relationship

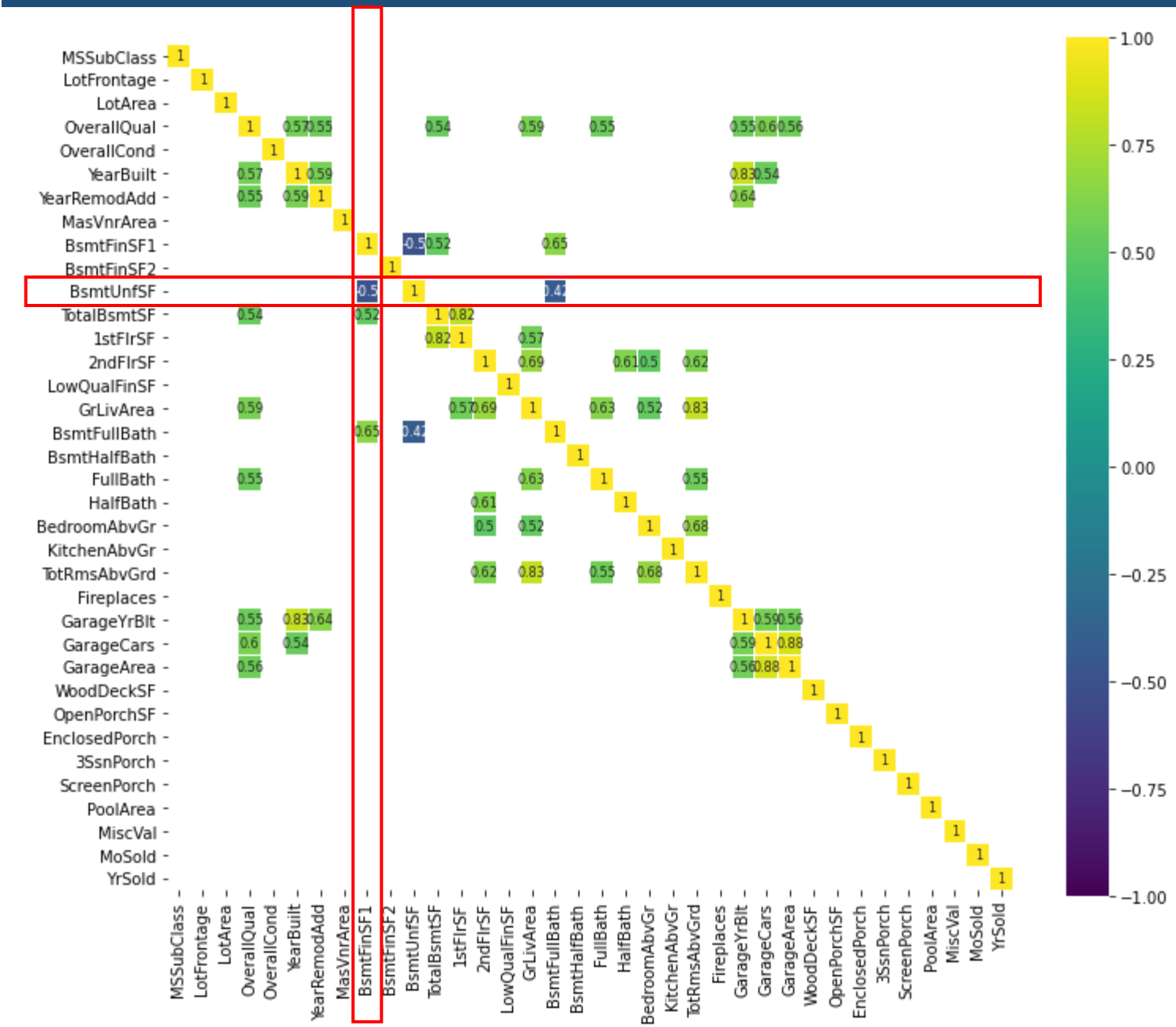

MSSubClass
LotFrontage
LotArea
OverallQual
OverallCond
YearBuilt
YearRemodAdd
MasVnrArea
BsmtFinSF1
BsmtFinSF2
BsmtUnfSF
TotalBsmtSF
1stFlrSF
2ndFlrSF
LowQualFinSF
GrLivArea
BsmtFullBath
BsmtHalfBath
FullBath
HalfBath
BedroomAbvGr
KitchenAbvGr
TotRmsAbvGrd
Fireplaces
GarageYrBlt
GarageCars
GarageArea
WoodDeckSF
OpenPorchSF
EnclosedPorch
3SsnPorch
ScreenPorch
PoolArea
MiscVal
MoSold
YrSold
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

FEATURE TO FEATURE RELATIONSHIP



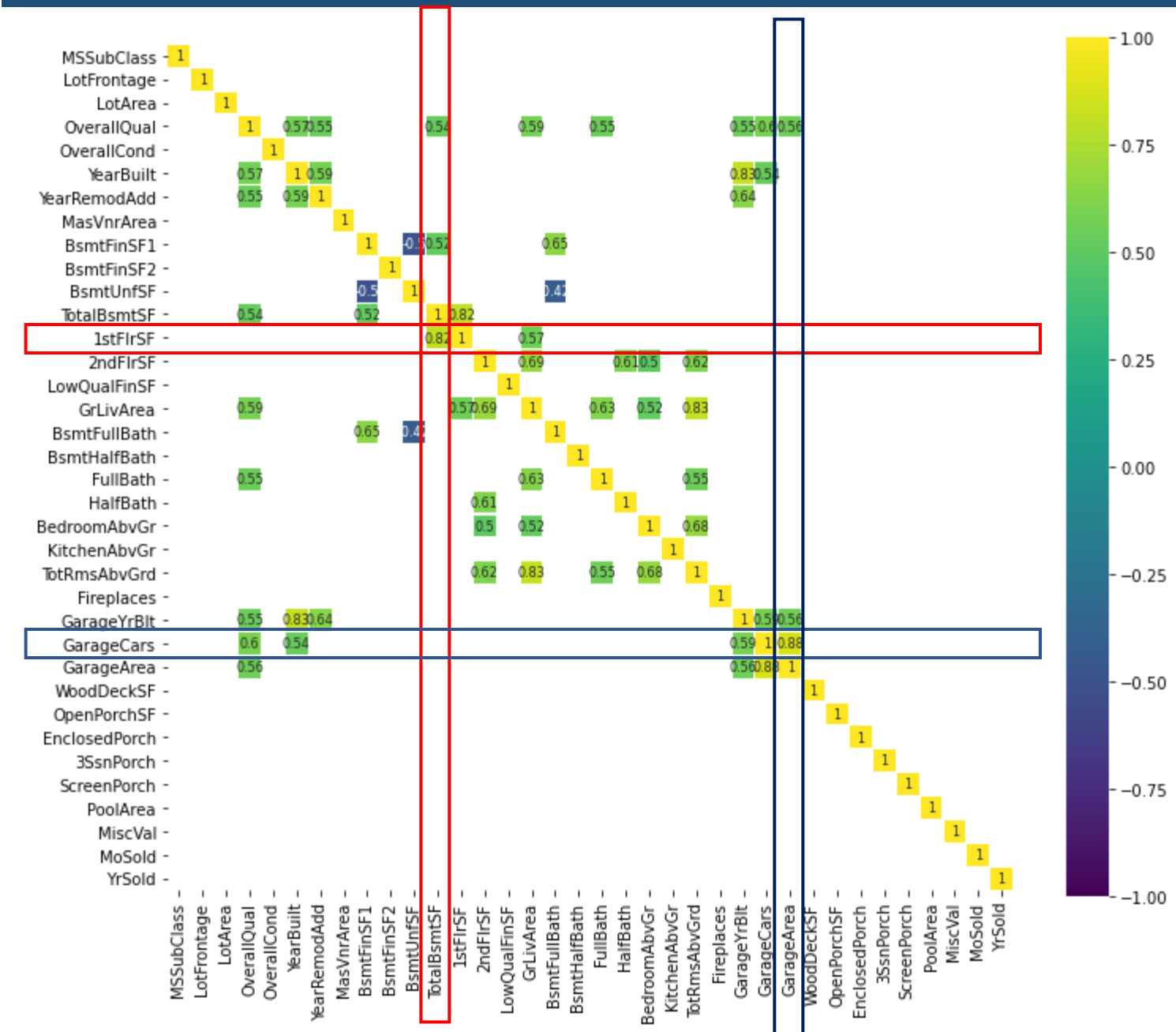
As for 1stFlrSF/TotalBsmtSF, it is normal that the more the 1st floor is large (considering many houses have only 1 floor), the more the total basement will be large.

FEATURE TO FEATURE RELATIONSHIP



There is a strong negative correlation between BsmtUnfSF (Unfinished square feet of basement area) and BsmtFinSF1 (Type 2 finished square feet).

FEATURE TO FEATURE RELATIONSHIP



We can conclude that, by essence, some of those features may be combined between each other in order to reduce the number of features (`1stFlrSF`/`TotalBsmtSF`, `GarageCars`/`GarageArea`) and others indicates that people expect multiples features to be packaged together.

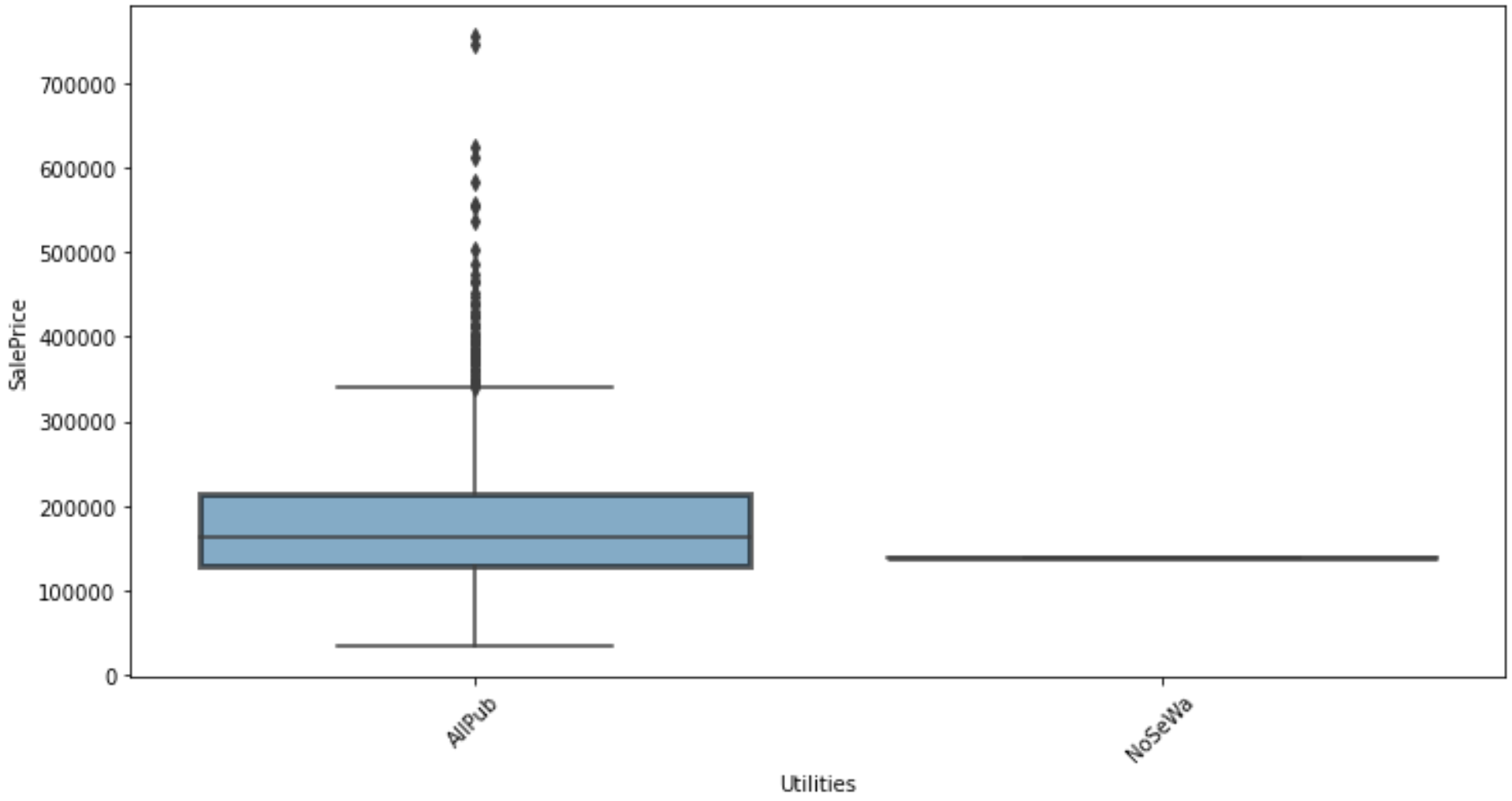
CATEGORICAL TO QUANTITAVIE RELATIONSHIP

Exercise:

1. Plot the box plot for , ‘Foundation’, ‘HeatingQC’, ‘Utilities’, ‘Heating’, ‘GarageCond’, `Functional`
2. Determine the categorical feature which has predominant category.

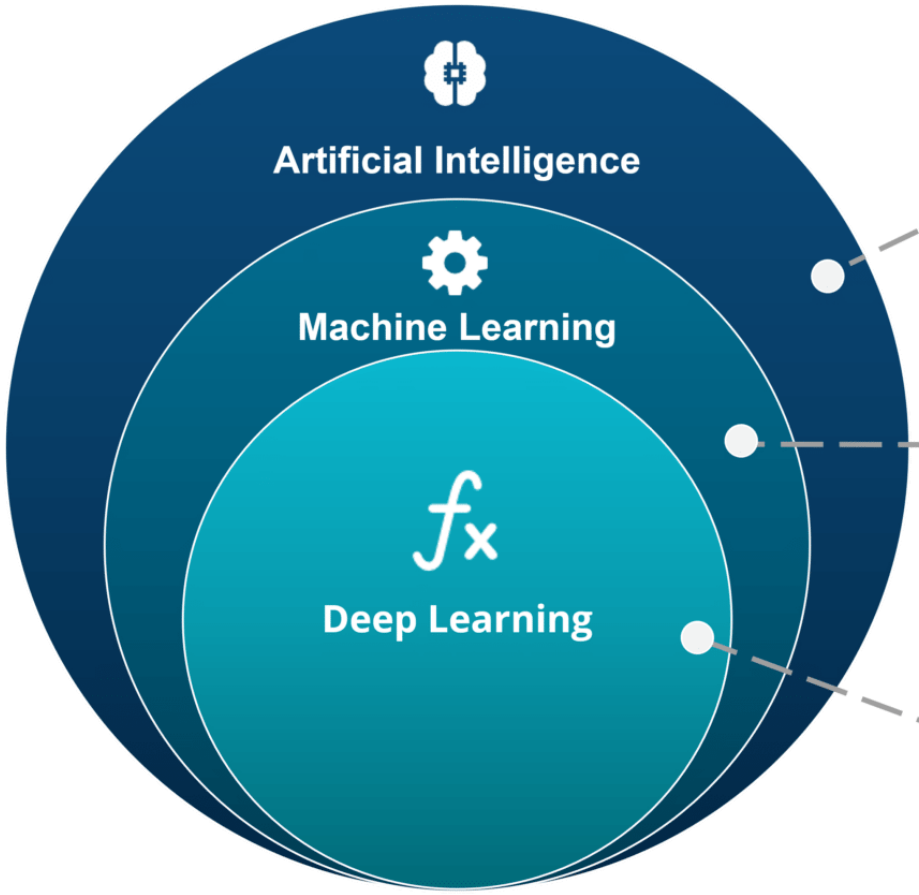
Note: Features with predominant category may not be relevant to build predictive model

CATEGORICAL TO QUANTITAVIE RELATIONSHIP



Note: Features with predominant category may not be relevant to build predictive model

DEEP LEARNING – NEURAL NETWORK



ARTIFICIAL INTELLIGENCE

A technique which enables machines to mimic human behaviour

MACHINE LEARNING

Subset of AI technique which use statistical methods to enable machines to improve with experience

DEEP LEARNING

Subset of ML which make the computation of multi-layer neural network feasible

DEEP LEARNING – NEURAL NETWORK

Most generally, a machine learning algorithm can be though of as a black box. It takes inputs and gives outputs.

The purpose of this course is to show you how to create this 'black box' and tailor it to your needs.

For example, we may create a model that predicts the weather tomorrow, based on meteorological data about the past few days.

The "black box" in fact is a mathematical model. The machine learning algorithm will follow a kind of trial-and-error method to determine the model that estimates the outputs, given inputs.

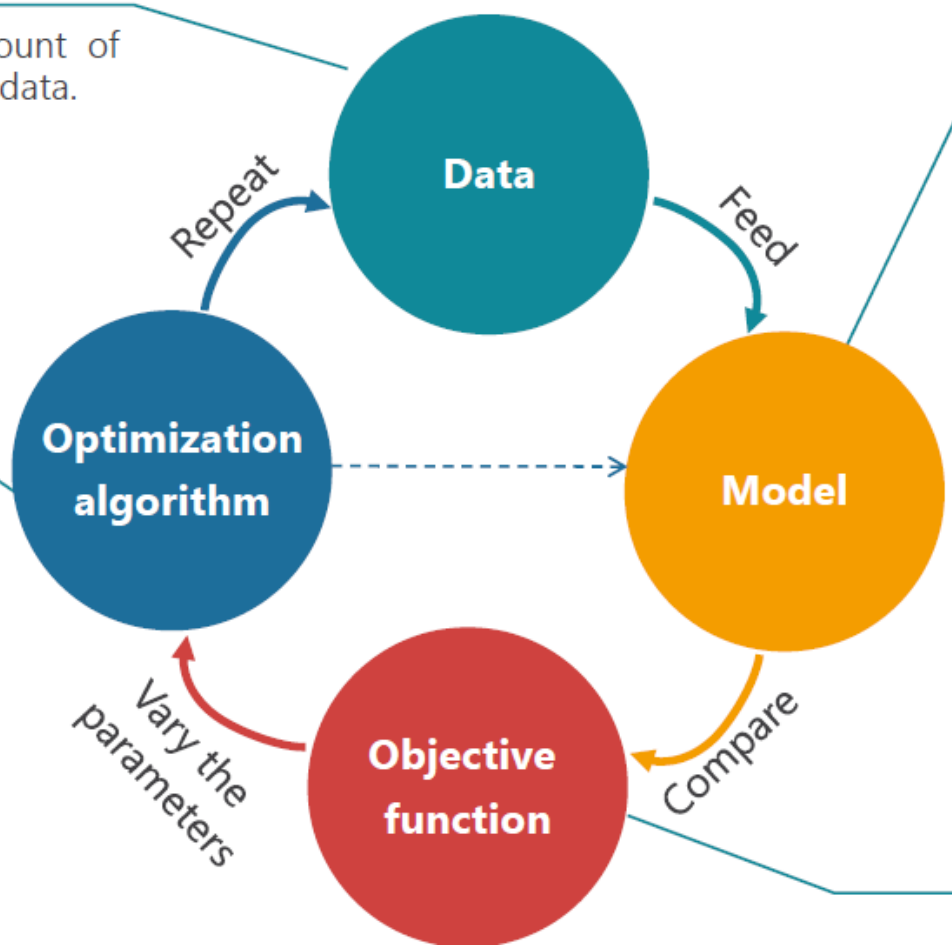
Once we have a model, we must **train** it. **Training** is the process through which, the model **learns** how to make sense of input data.

DEEP LEARNING – NEURAL NETWORK

The basic logic behind training an algorithm involves four ingredients: data, model, objective function, and optimization algorithm. They are **ingredients**, instead of steps, as the process is iterative.

First, we need to prepare a certain amount of **data** to train on. Usually, we take historical data.

We choose the type of **model**. Roughly speaking, this is some function, which is defined by the *weights* and the *biases*. We feed the input data into the model. Essentially, the idea of the machine learning algorithm is to find the *parameters* for which the model has the highest predictive power.



We achieve the optimization using an **optimization algorithm.** Using the value of the objective function, the optimization algorithm *varies the parameters* of the model. This operation is repeated until we find the values of the parameters, for which the objective function is optimal.

The **objective function** measures the predictive power of our model. Mathematically, the machine learning problem boils down to *optimizing* this function. For example, in the case of loss, we are trying to *minimize* it.

DEEP LEARNING – NEURAL NETWORK

The simplest possible model is **a linear model**. Despite appearing unrealistically simple, in the deep learning context, it is the basis of more complicated models.

There are four elements.

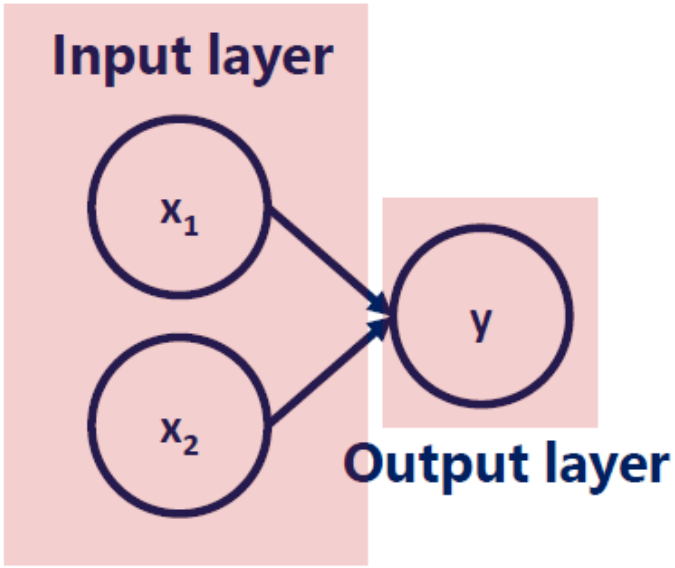
- The input(s), x. That's basically the data that we feed to the model.
- The weight(s), w. Together with the biases, they are called **parameters.** The optimization algorithm will vary the weights and the biases, in order to produce the model that fits the data best.
- The bias(es), b. See weight(s).
- The output(s), y. y is a function of x, determined by w and b.

Each model is determined solely by its parameters (the weights and the biases). That is why we are interested in varying them using the (kind of) trial-and-error method, until we find a model that explains the data sufficiently well.

DEEP LEARNING – NEURAL NETWORK

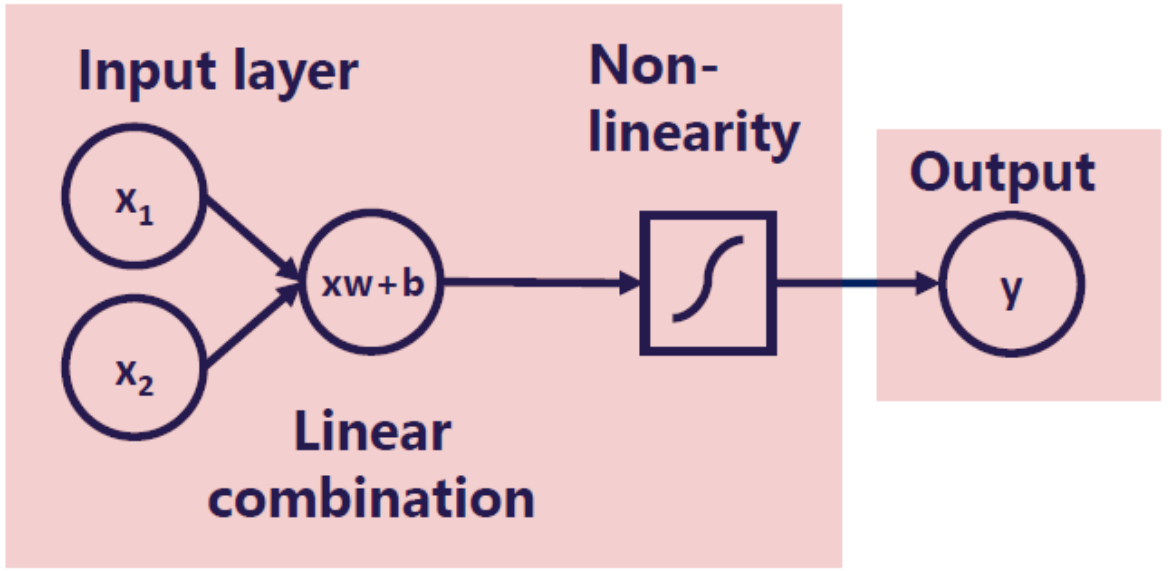
An initial linear combination and the added non-linearity form a **layer**. The layer is the building block of neural networks.

Minimal example (a simple neural network)



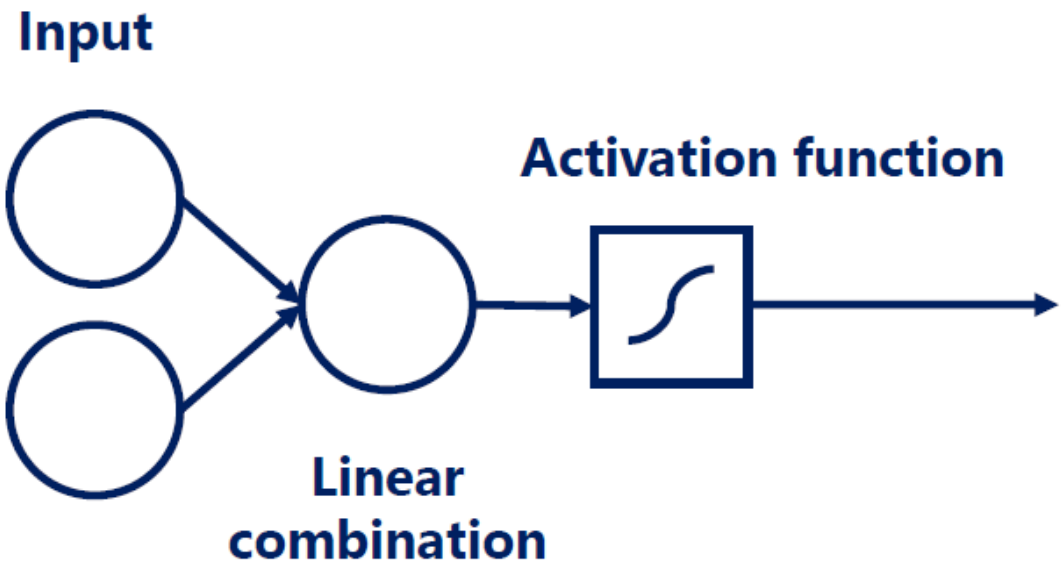
In the minimal example we trained a *neural network* which had no depth. There were solely an input layer and an output layer. Moreover, the output was simply **a linear combination** of the input.

Neural networks



Neural networks step on linear combinations, but add a non-linearity to each one of them. Mixing linear combinations and non-linearities allows us to model arbitrary functions.

DEEP LEARNING – NEURAL NETWORK



In the respective lesson, we gave an example of temperature change. The temperature starts decreasing (which is a numerical change). Our brain is a kind of an 'activation function'. It tells us whether it is **cold enough** for us to put on a jacket.

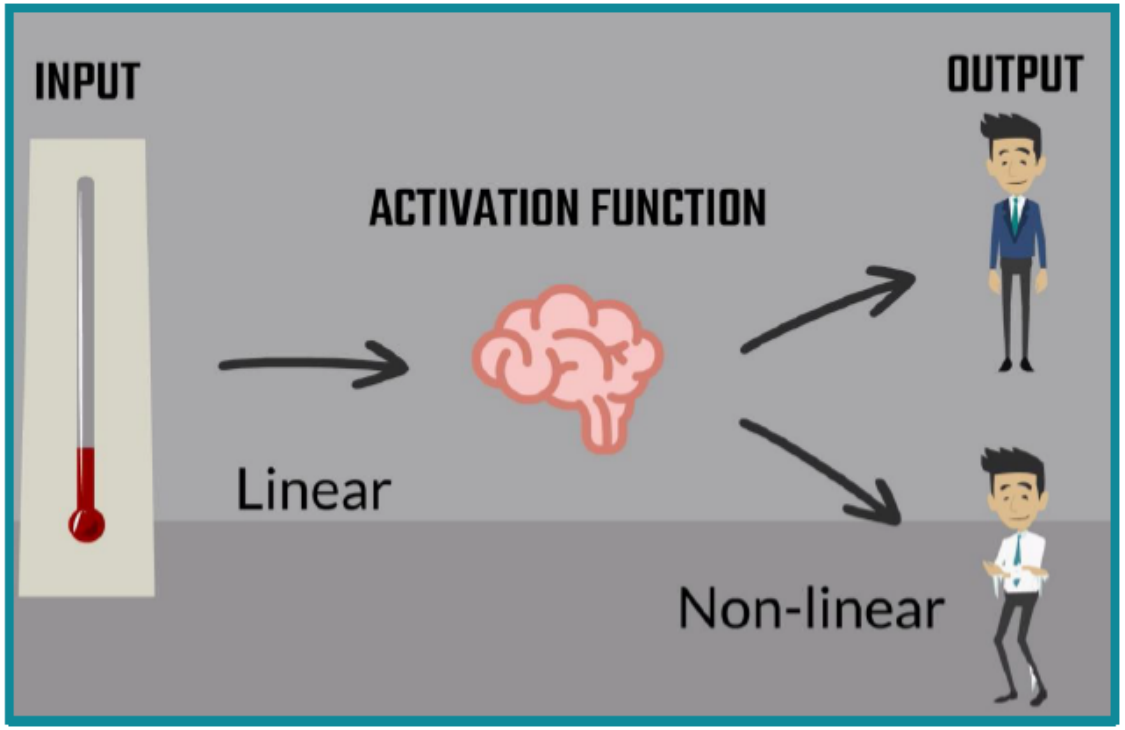
Putting on a jacket is a binary action: 0 (no jacket) or 1 (jacket).

This is a very intuitive and visual (yet not so practical) example of how activation functions work.

Activation functions (non-linearities) are needed so we can break the linearity and represent more complicated relationships.

Moreover, activation functions are required in order to **stack layers.**

Activation functions transform inputs into outputs of a different kind.



DEEP LEARNING – NEURAL NETWORK

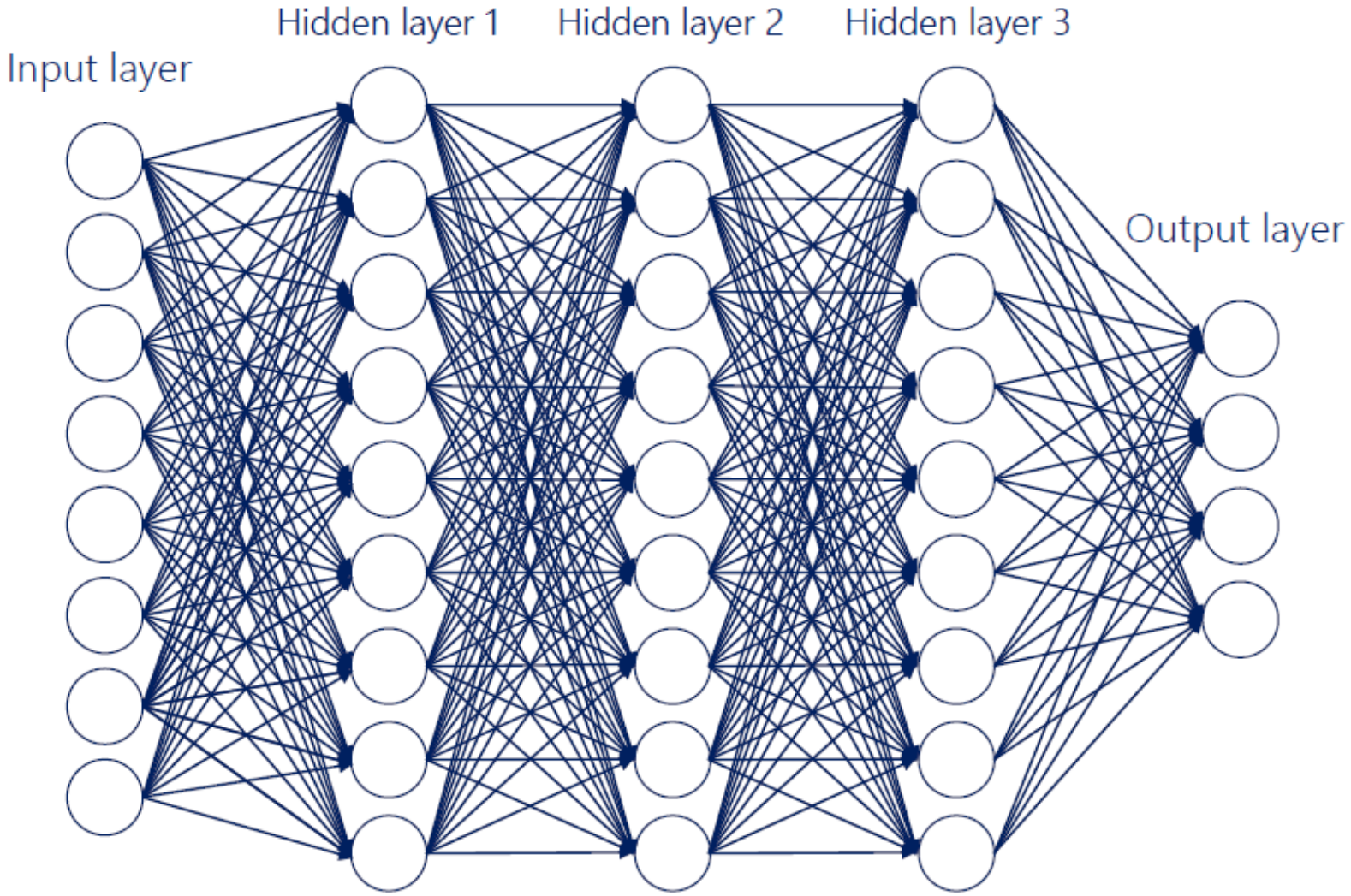
This is a deep neural network (deep net) with 5 layers.

How to read this diagram:

A layer

A unit (a neuron)

Arrows represent mathematical transformations



PRACTICAL

1. Import modules

```
import numpy as np
import matplotlib.pyplot as plt
```

Before getting started, we will need to import the necessary libraries. Only two libraries will be needed for this example, without plotting the loss we would only need Numpy. Numpy is a python math library mainly used for linear algebra applications. Matplotlib is a visualization tool that we will use to create a plot to display how our error decreases over time.

2. Generate data

```
inputs = np.array([[0, 1, 0],
                   [0, 1, 1],
                   [0, 0, 0],
                   [1, 0, 0],
                   [1, 1, 1],
                   [1, 0, 1]])

outputs = np.array([[0], [0], [0], [1], [1], [1]])
```

As mentioned earlier, neural networks need data to learn from. We will create our input data matrix and the corresponding outputs matrix with Numpy's .array() function. Each sample in the input consists of three feature columns made up of 0s and 1s that produce one output of either a 0 or 1. We want to neural network to learn that the outputs are determined by the first feature column in each sample.

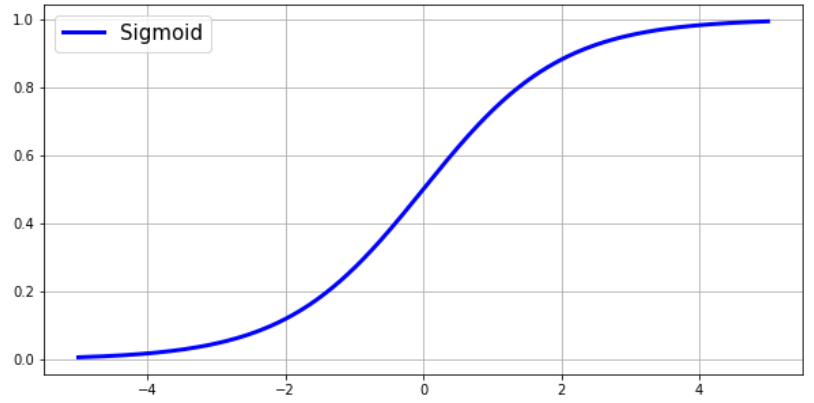
3. Create class

```
class NeuralNetwork:

    def __init__(self, inputs, outputs):
        self.inputs  = inputs
        self.outputs = outputs
        self.weights = np.array([[.50], [.50], [.50]])
        self.error_history = []
        self.epoch_list = []
```

We will take an object-oriented approach to build this particular neural network. We can begin by creating a class called "NeuralNetwork" and initializing the class by defining the `_init_` function. Our `_init_` function will take the inputs and outputs as arguments. We will also need to define our weights, which, for simplicity, will start with each weight being .50. Because each feature in the data must be connected to the hidden layer, we will need a weight for each feature in the data (three weights). For plotting purposes, we will also create two empty lists: loss_history and epoch_list. This will keep track of our neural network's error at each epoch during the training process.

4. Define sigmoid function

```
def sigmoid(self, x, deriv=False):
    if deriv == True:
        return x * (1 - x)
    return 1 / (1 + np.exp(-x))
```



This neural network will be using the sigmoid function, or logistic function, as the activation function. The sigmoid function is a popular nonlinear activation function that has a range of (0–1). The inputs to this function will always be squished down to fit in-between the sigmoid function's two horizontal asymptotes at $y=0$ and $y=1$. The sigmoid function has some well-known issues that restrict its usage. When we look at the graph below of the sigmoidal curve, we notice that as we reach the two ends of the curve, the derivatives of those points become very small. When these small derivatives are multiplied during backpropagation, they become smaller and smaller until becoming useless. Due to the derivatives, or gradients, getting smaller and smaller, the weights in the neural network will not be updated very much, if at all. This will lead the neural network to become stuck, with the situation becoming worse and worse for every additional training iteration.

The sigmoid function can be written as:

$$S(x) = \frac{1}{1 + e^{-x}}$$

And the derivative of the sigmoid function can be written as:

$S'(x) = S(x) \cdot (1 - S(x))$

5. Feed forward

```
def feed_forward(self):
    self.hidden = self.sigmoid(np.dot(self.inputs, self.weights))
```

During our neural network's training process, the input data will be fed forward through the network's weights and functions. The result of this feed-forward function will be the output of the hidden layer or the hidden layer's best guess with the weights it is given. Each feature in the input data will have its own weight for its connection to the hidden layer. We will start by taking the sum of every feature multiplied by its corresponding weight. Once we have multiplied the input and weight matrices, we can take the results and feed it through the sigmoid function to get squished down into a probability between (0–1). The forward propagation function can be written like this, where x_i and w_i are individual features and weights in the matrices:

Input Matrix | Weights | ∑(Input x Weight)

X | w | ∑xw | ∑xw

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} .50 \\ .50 \\ .50 \end{bmatrix} = \begin{bmatrix} (0 \times .50) + (0 \times .50) + (1 \times .50) \\ (0 \times .50) + (1 \times .50) + (0 \times .50) \\ (1 \times .50) + (0 \times .50) + (1 \times .50) \\ (1 \times .50) + (1 \times .50) + (0 \times .50) \end{bmatrix} = \begin{bmatrix} .50 \\ .50 \\ 1 \\ 1 \end{bmatrix}$$

6. Back propagation

```
def backpropagation(self):
    self.error  = self.outputs - self.hidden
    delta = self.error * self.sigmoid(self.hidden, deriv=True)
    self.weights += np.dot(self.inputs.T, delta)
```

This is the coolest part of the whole neural net: backpropagation. Backpropagation will go back through the layer(s) of the neural network, determine which weights contributed to the output and the error, then change the weights based on the gradient of the hidden layers output. This will be explained further, but for now, the whole process can be written like this, where y is the correct output and $\hat{y}$ is the hidden layers prediction:

$$w_i + X^T \cdot (y - \hat{y}) \cdot \frac{1}{1 + e^{-(\sum x_i w_i)}} \cdot (1 - \frac{1}{1 + e^{-(\sum x_i w_i)}})$$

To calculate the error of the hidden layer's predictions, we will simply take the difference between the correct output matrix, *y*, and the hidden layer's matrix, $\hat{y}$.

$$\underset{y}{\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}} - \underset{\hat{y}}{\begin{bmatrix} .62 \\ .62 \\ .73 \\ .73 \end{bmatrix}} = \underset{\text{Error}\ (y - \hat{y})}{\begin{bmatrix} -.62 \\ -.62 \\ .27 \\ .27 \end{bmatrix}}$$

7. Train the model

```
def train(self, epochs=25000):
    for epoch in range(epochs):
        self.feed_forward()
        self.backpropagation()

        self.error_history.append(np.average(np.abs(self.error)))
        self.epoch_list.append(epoch)
```

The time has come to train the neural network. During the training process, the neural net will "learn" which features in the input data correlate with its output, and it will learn to make accurate predictions. To train our neural network, we will create the train function with the number of epochs, or iterations to 25,000. This means the neural network will repeat the weight-updating process 25,000 times. Within the train function, we will call our `feed_forward()` function, then the `backpropagation()` function. For each iteration, we will also keep track of the error produced after the `feed_forward()` function has completed. We will keep track of this by appending the error and epoch to the lists that were initialized earlier. I am sure there is an easier way to do this, but for quick prototyping, this way works just fine for now.

The training process follows the equation below for every weight in our neural net:

x_i — Feature in Input Data

w_i — The Weight that is Being Updated

X^T — Transposed Input Data

y — Correct Output

$\hat{y}$ — Predicted Output

$(y - \hat{y})$ — Error

$\sum x_i w_i$ — Sum of the Products of Input Features and Weights

$S(\sum x_i w_i)$ — Sigmoid Function

$$w_i + X^T \cdot (y - \hat{y}) \cdot S(\sum x_i w_i)(1 - S(\sum x_i w_i))$$

8. Predict

```
def predict(self, new_input):
    prediction = self.sigmoid(np.dot(new_input, self.weights))
    return prediction
```

Now that the neural network has been trained and has learned the important features in the input data, we can begin to make predictions. The prediction function will look similar to the hidden layer, or the feed_forward() function. The forward propagation function essentially makes a prediction as well, then backpropagation checks for the error and updates the weights. Our predict function will use the same method as the feedforward function: multiply the input matrix and the weights matrix, then feed the results through the sigmoid function to return a value between 0-1. Hopefully, our neural network will make a prediction as close as possible to the actual output.

```
NN = NeuralNetwork(inputs, outputs)
```

We will create our NN object from the NeuralNetwork class and pass in the input matrix and the output matrix.

```
NN.train()
```

Now we can create the two new examples that we want our neural network to make predictions for. We will call these “example” and “example_2”. We can then call the .predict() function and pass through the arrays. We know that the first number, or feature, in the input determines the output. The first example, “Example”, has a 1 in the first column, and therefore the output should be a one. The second example has a 0 in the first column, and so the output should be a 0.

We can then call the `.train()` function on our neural network object.

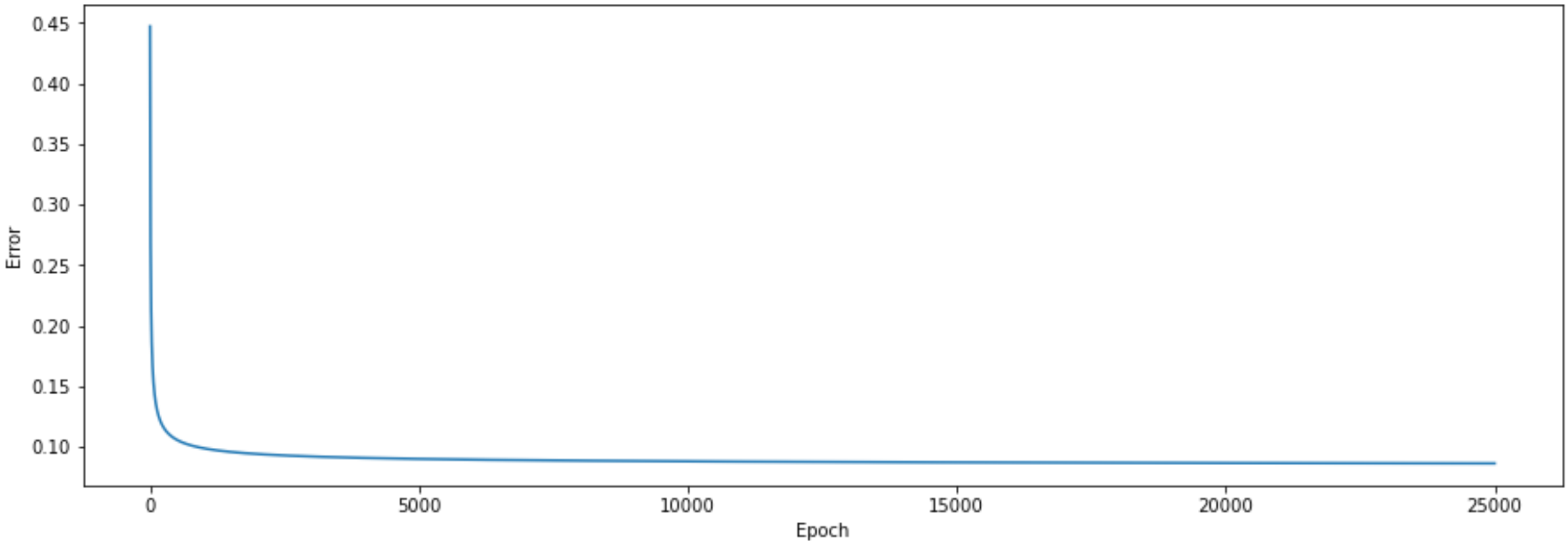
```
example = np.array([[1, 1, 0]])
example_2 = np.array([[0, 1, 1]])

print(NN.predict(example), ' - Correct: ', example[0][0])
print(NN.predict(example_2), ' - Correct: ', example_2[0][0])
```

Output

```
[[0.99089925]]  - Correct:  1
[[0.006409]]  - Correct:  0
```

```
plt.figure(figsize=(15,5))
plt.plot(NN.epoch_list, NN.error_history)
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

With the training complete, we can plot the error over each training iteration. The plot shows that there is a huge decrease in error during the earlier epochs, but that the error slightly plateaus after approximately 5000 iterations.


0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
Error
0
5000
10000
15000
20000
25000
Epoch