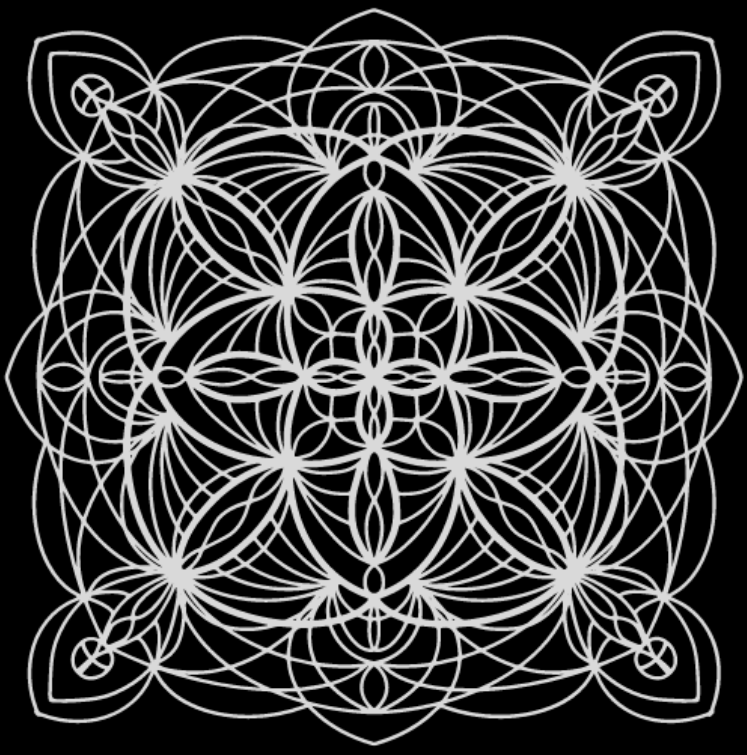
Murat Durmus

# A PRIMER TO THE 42 MOST COMMONLY USED MACHINE LEARNING ALGORITHMS

***

(With Code Samples)



## Cover design:

Murat Durmus (Mandala: ilona illustrations - canva.com)

## About the Author

Murat Durmus is CEO and founder of AISOMA (a Frankfurt am Main (Germany) based company specializing in AI-based technology development and consulting) and Author of the books "Mindful AI - Reflections on Artificial Intelligence".& "INSIDE ALAN TURING"

You can get in touch with the author via:

- LinkedIn: https://www.linkedin.com/in/ceosaisoma/

- E-Mail: murat.durmus@aisoma.de

Note:

The code examples and their description in this book were written with the support of ChatGPT (OpenAI).

*“All models are wrong,*

*but some are useful.”*

George E. P. Box

# INTRODUCTION

Machine learning refers to the development of AI systems that can perform tasks due to a "learning process" based on data. This is in contrast to approaches and methods in symbolic AI and traditional software development, which are based on embedding explicit rules and logical statements in the code. ML is at the heart of recent advances in statistical AI and the methodology behind technological achievements such as computer programs that outperform humans in tasks ranging from medical diagnosis to complex games. The recent surge of interest in AI is largely due to the achievements made possible by ML. As the term "statistical AI" suggests, ML draws on statistics and probability theory concepts. Many forms of ML go beyond traditional statistical methods, which is why we often think of ML as an exciting new field. However, despite the hype surrounding this technological development, the line between ML and statistics is blurred. There are contexts in which ML is best viewed as a continuum with traditional statistical methods rather than a clearly defined separate field. Regardless of the definitional boundaries, ML is often used for the same analytical tasks that conventional statistical methods have been used for in the past. ML Approaches.

ML is a very active area of research that encompasses a broad and ever-evolving range of methods. Three primary approaches can be distinguished at a high level: **supervised learning**, **unsupervised learning**, and r**einforcement learning**.

**Supervised Learning**

In supervised learning, the task of the ML algorithm is to infer the value of a predefined target variable (or output variable) based on known values of feature variables (or input variables). The

presence of labeled data (i.e., data with known values for the target in question) is a prerequisite for supervised learning. The learning process consists of developing a model of the relationship between feature and target variables based on labeled training data. This process is also referred to as "model training." After a successful training phase (which is confirmed by a testing phase also based on labeled data), the resulting model can be applied to unlabeled data to infer the most likely value of the target variable. This is referred to as the inference phase.

Supervised learning can solve two main types of analytic problems:

- **Regression problems** where the target variable of interest is continuous. Examples include predicting future stock prices or insurance costs.
- **Classification problems**, where the target of interest is a categorical variable. These include issues where the target variable is binary (e.g., whether a financial transaction is fraudulent or non-fraudulent) and multi-class problems that involve more than two categories. For example, classification can be used to assess the likelihood that customers will default on loan repayments.

**Unsupervised Learning**

Unsupervised learning involves identifying patterns and relationships in data without a predefined relationship of interest. Unlike supervised learning, this approach does not rely on labeled training data. Therefore, unsupervised learning can be more exploratory, although the results are not necessarily less meaningful.

***Unsupervised learning is beneficial when labeled data is unavailable or expensive to produce. This approach can be used to solve problems such as the following:***

**Cluster analysis** involves grouping units of observations based on similarities and dissimilarities between them. Examples of tasks where cluster analysis can be helpful include customer segmentation exercises.

**Association analysis**, where the goal is to identify salient relationships among variables within a data set. Association rules (i.e., formal if-then statements) typically describe such relationships. These rules can lead to findings such as "customers interested in X are also interested in Y and Z." Association analysis is used for product recommendation and customer service management tasks.

**Reinforcement Learning**

Reinforcement learning is based on the concept of an "agent" exploring an environment. The agent's task is to determine an optimal action or sequence of steps (the goal of interest) in response to its environment. The learning process does not rely on examples of "correct responses." Instead, it depends on a reward function that provides feedback on the actions taken. The agent strives to maximize its reward and thus improve its performance through an iterative process of trial and error.

Reinforcement learning is practical when the optimal actions (i.e., the correct responses) are unknown. In such situations, labeled training data are not available or risk producing suboptimal results when analysts use supervised learning. The conceptual structure of the approach also makes it relevant for problem types that have a sequential or dynamic nature. Examples include problems in robotics or games.

Much work on reinforcement learning is taking place in the context of basic research. This includes research in general AI. Compared to other ML approaches, reinforcement learning is less common in business. The most noted business applications are outside of financial services and include autonomous vehicles and other forms of robotics. Potential applications in financial services include trading or trade execution and dynamic pricing.

These three approaches include a variety of ML methods such as linear regression, decision trees, support vector machines, artificial neural networks, and ensemble methods. However, two general points about methodological differences are worth noting.

First, ML methods differ significantly in complexity. Discussions of ML often focus on practices with a high degree of complexity. For example, neural networks, a family of techniques that search for patterns and relationships in data sets using network structures similar to those found in the biological brain, receive considerable attention. However, ML also includes fewer complex methods such as ordinary least squares regression and logistic regression. These more straightforward methods have long been used in statistics and econometrics and were established before ML emerged in its current form. We will return to the issue of complexity and its practical implications in later chapters. It should be noted that ML as a field encompasses specific, highly complex methods but is not limited to them.

Second, ML methods can be used to design static or dynamic systems. For static systems, ML is used to develop models that do not evolve once they are deployed unless a new model intentionally replaces them. In dynamic systems, on the other hand, models continue to adapt after deployment based on new data that becomes available during operation.

Such dynamic (or incremental) learning can greatly benefit situations where the data available during development is limited or where models capture phenomena with rapidly changing characteristics.

# The Taxonomy used in this book

***Main Domain and Data Types***

| Main Domain | Data Type | Definition |
|---|---|---|
| **Computer Vision** | Image | Visual representation of a pixel matrix consisting of one channel for black and white images, three elements for color images (RGB), or four elements for color images with opacity (RGBA). |
| | Video | A succession of images (frames), sometimes grouped with a time series (a sound). |
| **NLP / Speech Processing** | Text | A succession of characters (e.g., a tweet, a text field). |
| | Time Series | A series of data points (e.g., numerical) indexed in time order. |
| **Classic Data Science** | Structured Data | Data is organized in a predefined array model with a specific column for each characteristic (e.g., text, numeric data, date). To be more precise, structured data refers to organized data found, for example, in a relational database (which, as mentioned, may contain columns of text).<br><br>Quantitative data can be distinguished from qualitative data. Quantitative data correspond to numeric data that can support some arithmetic operations, while qualitative data are usually used as categorical data to classify data according to their similarities. |

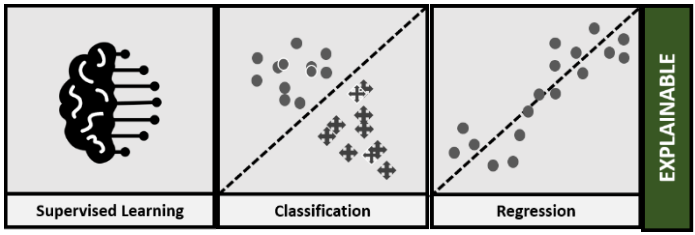
***Learning paradigms with subtypes.***

| Learning Paradigm | Subtype | Definition |
|---|---|---|
| **Supervised Learning** | Classification | Classification is the process of predicting the class of given data points. (Is the picture a cat or a dog?) |
| | Regression | Regression models are used to predict a continuous value. (Predict the price of a house based on its features). |
| **Unsupervised Learning** | Clustering | Clustering is the task of dividing data points into multiple groups so that data points in the same groups are more similar to each other than the data points in the other groups. |
| | Dimensionality Reduction | Dimensionality reduction refers to techniques for reducing the number of input variables in the training data. |
| **Reinforcement Learning** | Rewarding | The reward is an area of ML that deals with how intelligent agents should act in an environment to maximize the notion of cumulative reward by learning from their experiences through feedback. |

***Explainability***

An important aspect of AI security is explainability. Understanding the algorithms and making them explainable makes them accessible to as many people as possible. In addition, explainability helps increase the trustworthiness of AI and supports forensics and analysis of decisions.

# ADABOOST




| | |
|---|---|
| **Definition** | **AdaBoost uses multiple iterations to create a single composite strong learner by iteratively adding weak learners. In each training phase, a new weak learner is added to the ensemble and a weight vector is adjusted to focus on examples that were misclassified in previous rounds.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification, Regression** |
| **Explainability** | **Explainable** |

AdaBoost (Adaptive Boosting) is an ensemble learning algorithm used to improve the accuracy of weak classifiers by combining them into a strong classifier. A classifier is a model that can predict the class or category of input, and a weak classifier is a model that performs better than random guessing but not as well as a strong classifier.

The AdaBoost algorithm works by iteratively training a series of weak classifiers on the data and adjusting the weights of the samples in training set at each iteration. The algorithm assigns higher weights to the samples misclassified by the previous classifiers and lower weights to the samples correctly classified. This process is repeated for a fixed number of iterations or until a stopping criterion is met.

At the end of the process, the algorithm combines the outputs of all the weak classifiers into a final strong classifier. The combination is done by assigning a weight to each weak classifier based on its accuracy. The last strong classifier assigns a class or category to the input by taking a weighted majority vote of the outputs of all the weak classifiers.

AdaBoost is a powerful algorithm that has been used in various applications, including image and speech recognition, object detection, and bioinformatics. It is beneficial when the data is noisy or has multiple features and is resistant to overfitting.

One of the main advantages of AdaBoost is that it can be used with a variety of weak classifiers, including decision trees, neural networks, and support vector machines. It's also simple to implement and computationally efficient. However, it is sensitive to outliers and noise in the data, and it can be affected by choice of weak classifier and the number of iterations.

Example:

Imagine we have a dataset with 100 observations, each with two features (x1 and x2) and a binary label (1 or -1). We want to train a classifier that can predict the label of a new observation based on its features.

1. The algorithm starts by training a weak classifier on the data, for example, a decision stump (a one-level decision tree) that splits the data based on a threshold value of one of the features. This classifier correctly classifies 80 of the observations.
2. Next, the algorithm assigns a weight to each observation based on whether it was correctly or incorrectly classified. The weight of the correctly classified observations is reduced, and the weight of the incorrectly classified observations is increased.
3. The algorithm then trains a second weak classifier on the data using the updated weights. This classifier may be different from the first one; for example, it could use an additional feature or another threshold value. This classifier correctly classifies 85 of the observations.
4. The algorithm assigns new weights to the observations and repeats the process for a fixed number of iterations or until a stopping criterion is met.
5. At the end of the process, the algorithm has trained several weak classifiers on the data, assigning a weight to each classifier based on its accuracy. The final strong classifier is a weighted majority vote of the outputs of all the weak classifiers.

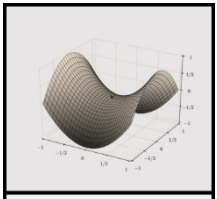
An example of how to use the Adaboost algorithm in Python using the scikit-learn library:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import make_classification

# Generate some example data
X, y = make_classification(n_features=4, n_informative=2,
                           n_redundant=0, random_state=0)

# Create an instance of the Adaboost classifier
clf = AdaBoostClassifier(random_state=0)

# Fit the model to the data
clf.fit(X, y)

# Make predictions on new data
predictions = clf.predict(X)
```

In this example, we first import the **AdaBoostClassifier** class from the **ensemble** module of scikit-learn. Then, we use the **make_classification** function to generate example data for the model. Next, we create an instance of the classifier, setting the random state to 0 for reproducibility. Then, we use the **fit** method to train the model on the data and the **predict** method to make predictions on new data.

It's worth noting that the **AdaBoostClassifier** can be used for classification problems. If you want to use Adaboost for regression, you can use the **AdaBoostRegressor** class instead.

# ADAM OPTIMIZATION



Optimization

| | |
|---|---|
| **Definition** | **Adam optimization is an extension of stochastic gradient descent. It can be used instead of classical stochastic gradient descent to update the network weights more efficiently thanks to two methods: adaptive learning rate and momentum.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **-** |
| **Learning Paradigm** | **Optimization** |
| **Explainability** | **-** |

Adam (Adaptive Moment Estimation) is an optimization algorithm used to update the parameters of a machine learning model during training. It is a popular algorithm used in deep learning and neural networks.

Adam is an extension of the stochastic gradient descent (SGD) algorithm, which is a method to optimize the parameters of a model by updating them in the direction of the negative gradient of the loss function. The Adam algorithm, like SGD, uses the gradients of the loss function concerning the model parameters to update the parameters. In addition, it also incorporates the concept of "momentum" and "adaptive learning rates" to improve the optimization process.

The "momentum" term in Adam is similar to the momentum term used in other optimization algorithms like SGD with momentum. It helps the optimizer to "remember" the direction of the previous update and continue moving in that direction, which can help the optimizer to converge faster.

The "adaptive learning rates" term in Adam adapts the learning rate for each parameter based on the historical gradient information. This allows the optimizer to adjust the learning rate for each parameter individually so that the optimizer can converge faster and with more stability.

Adam is widely used in deep learning because it is computationally efficient and can handle sparse gradients and noisy optimization landscapes. But it requires more memory to store the historical gradient information, and it may be sensitive to the choice of hyperparameters, such as the initial learning rate.

In summary, Adam is a powerful optimization algorithm that can improve the convergence speed and stability of the model during training by incorporating the concepts of momentum and

adaptive learning rates; it's widely used in deep learning and neural networks as it is computationally efficient and can handle noisy optimization landscapes.

Example:

Imagine we have a neural network with two layers, the first layer has four neurons and the second layer has one neuron; the network is used for binary classification. The goal is to find the optimal values for the weights and biases of the neurons that minimize the loss function.

1. The algorithm starts by initializing the weights and biases of the neurons randomly.
2. Next, the algorithm performs a forward pass of the data through the network to calculate the output of the neurons and the loss function.
3. The algorithm then calculates the loss function's gradients for the neurons' weights and biases.
4. The algorithm uses the gradients to update the weights and biases of the neurons using Adam optimization. The update step includes calculating the moving average of the gradients and the squared gradients, which are used to adjust the learning rate for each weight and bias individually.
5. The algorithm repeats steps 2-4 for a fixed number of iterations or until a stopping criterion is met.
6. At the end of the process, the algorithm has found the optimal values for the weights and biases of the neurons that minimize the loss function.

The example I provided is a simplified version of the process; in practice, the neural network may have more layers and neurons, and the dataset may be much more significant. Also, the example

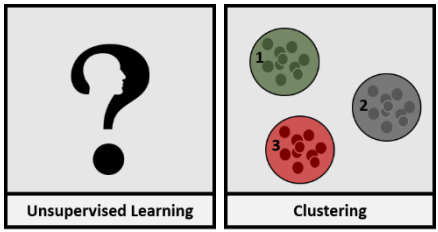
shows the process for a binary classification problem, but the Adam optimization algorithm can be used to optimize any differentiable loss function.

Code example of how to use the Adam optimization algorithm in Python with the Keras library:

```
from keras.optimizers import Adam

# Create a model
model = ...

# Compile the model with Adam optimizer
opt = Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
epsilon=None, decay=0.0, amsgrad=False)
model.compile(optimizer=opt, loss='binary_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

In this example, the Adam optimizer is being used with the specified learning rate (lr) and beta values, and the model is being compiled with binary crossentropy loss and accuracy metrics. The model is then trained on X_train and y_train data with 10 epochs and a batch size of 32. It's worth noting that this is just one of many ways to code Adam Optimization in Keras and the specific hyperparameter values can be adjusted based on the dataset and the problem at hand.

# AGGLOMERATIVE CLUSTERING




| | |
|---|---|
| **Definition** | **Agglomerative clustering is a "bottom-up" approach to hierarchical clustering. Each observation starts in its cluster, and cluster pairs are merged as they move up the hierarchy.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

Agglomerative Clustering is a type of hierarchical clustering algorithm. Hierarchical clustering algorithms are a class of algorithms that create a hierarchy of clusters, where each cluster is a subset of the previous one. In contrast, other clustering algorithms like k-means make flat clusters where each point belongs to exactly one cluster.

Agglomerative Clustering starts with each point as an individual cluster, then iteratively merges the closest pair of clusters until all points belong to a single cluster or a stopping criterion is met. The main idea behind agglomerative Clustering is that similar topics are more likely to be in the same cluster, and therefore, the algorithm starts with a large number of small clusters and ends with a small number of large clusters.

One of the critical parameters in Agglomerative Clustering is the linkage criteria, which determines the distance between clusters. Common linkage criteria include:

- Single linkage (the distance between the closest points in each cluster).
- Complete connection (the distance between the farthest points in each cluster).
- Average link (the average distance between all points in each cluster).
- Ward linkage (the minimum variance of distances between all points in each cluster).

Agglomerative Clustering is an efficient and flexible algorithm for clustering data. However, it has some limitations. For example, it does not scale well to large datasets, is sensitive to the linkage criteria, and needs to provide a way to determine the optimal number of clusters. Despite these limitations, it's a widely used

algorithm, and it is used in many applications such as image analysis, bioinformatics, and customer segmentation.

Example:

Imagine we have a dataset with 6 points, represented by the coordinates (x, y) in a two-dimensional space: (1,2), (2,4), (3,5), (4,4), (5,2), (6,1)

1. The algorithm starts by treating each point as an individual cluster. So, we have 6 clusters, each containing one point.
2. Next, the algorithm finds the closest pair of clusters and merges them into a new cluster. The linkage criteria used in this example is "single linkage," which means the algorithm finds the minimum distance between the closest points of each cluster. For example, the closest pair of clusters is (1,2) and (6,1), with a distance of 1.
3. The process is repeated, and the algorithm finds the next closest pair of clusters. In this example, the closest pair is (2,4) and (5,2), with a distance of 2.
4. The algorithm continues to merge clusters until all points are in the same cluster. In this case, the final cluster contains all 6 points and forms a triangle shape.

In this example, the number of clusters is determined by the stopping criterion, which is merging all the points in one cluster. However, in practice, one can use other stopping criteria, such as a maximum number of clusters or a threshold for the linkage distance.

Keep in mind that this is a simple example, and the process can be different depending on the linkage criteria, stopping criteria, and the shape of the data. Also, this example uses a 2-dimensional

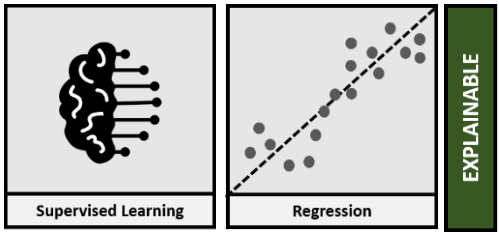
space, but the algorithm can work with any dimensions, and the linkage criteria can be adjusted accordingly.

Code example of how to use the scikit-learn library to perform agglomerative clustering in Python:

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs

# Create some sample data
X, y = make_blobs(n_samples=100, centers=3,
random_state=42)

# Initialize the agglomerative clustering model
agg_clustering = AgglomerativeClustering(n_clusters=3)

# Fit the model to the data
agg_clustering.fit(X)

# Predict the cluster labels for each data point
agg_clustering_labels = agg_clustering.labels_

print(agg_clustering_labels)
```

In this example, we first generate a sample dataset of 100 points in 3 clusters using the make_blobs function from the sklearn.datasets module. Then, we initialize an AgglomerativeClustering object with 3 clusters and fit it to the data using the fit() method. Finally, we predict the cluster labels for each data point using the labels_ attribute of the fitted model.

# ARMA/ARIMA MODEL




| | |
|---|---|
| **Definition** | **Given a time series Xt, the ARMA/ARIMA model is a tool for understanding and predicting the future values of this series. The model consists of an autoregressive part (AR) and a moving average part (MA).** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Time Series** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Regression** |
| **Explainability** | **Explainable** |

ARMA (AutoRegressive Moving Average) and ARIMA (AutoRegressive Integrated Moving Average) are time series forecasting models used to analyze and forecast univariate time series data. They are widely used in finance, economics, and other fields to analyze and predict trends, patterns, and fluctuations in data over time.

ARMA models are a combination of two types of models:

AutoRegressive (AR) models are based on the idea that past values of a time series can be used to predict future values. The model uses a linear combination of past observations to predict the current observation.

Moving Average (MA) models are based on the idea that the current observation is a weighted average of past errors or residuals. The model uses a linear combination of past errors to predict the recent observation.

ARIMA models are an extension of ARMA models, which include the concept of differencing to handle non-stationary time series data. Non-stationary data is characterized by a trend or a seasonality, which makes it challenging to model and forecast.

Differencing is transforming the data by subtracting the previous observation from the current observation. The differenced data becomes stationary and can be modeled with ARMA models.

The notation for an ARIMA model is (p,d,q), where:

- p is the number of auto-regressive terms (AR)
- d is the number of differences needed to make the data stationary (I for integrated)
- q is the number of moving average terms (MA)

For example, an ARIMA(1,1,1) model would include one auto-regressive term, one differencing term, and 1 moving average term.

ARIMA models are effective in forecasting time series data, but they have some.

Example:

Imagine we have a dataset that contains the monthly sales of a retail store for the past 36 months. The goal is to use the ARIMA model to forecast sales for the next 12 months.

1. The first step is to plot the data and analyze it to check if it has a trend or seasonality. The plot shows that the data has a clear upward trend, indicating that it is non-stationary.
2. Next, we need to make the data stationary by differencing it. We can do this by subtracting the previous month's sales from the current month's sales. This will remove the trend and make the data more predictable.
3. After making the data stationary, we will use the ACF(Auto-correlation function) and PACF(Partial Auto-correlation function) plots to identify the number of Auto-regressive (p) and Moving average (q) terms.
4. After identifying the p and q terms, we will now build the ARIMA model using (p, d, q) notation, where d is the number of differences required to make the data stationary. In this case, d = 1 as we have differenced the data once.
5. Once the model is built, we can use it to make predictions for the next 12 months.

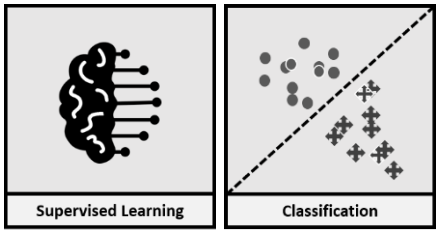
It's important to note that this is a simplified example; in practice, the process of building an ARIMA model is more complex and may require several iterations to find the optimal values for p, d, and q and other parameters such as the order of differencing and selecting the appropriate model. Additionally, different techniques can be used to identify the optimal parameters and validate the model, such as the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC).

Code example of ARIMA model implementation in Python using the statsmodels library:

```
import statsmodels.api as sm
from statsmodels.tsa.arima_model import ARIMA

# Load data
data = sm.datasets.sunspots.load_pandas().data

# Fit the model
arima_model = ARIMA(data['SUNACTIVITY'], order=(1, 1, 1))
arima_fit = arima_model.fit()

# Print summary of fit model
print(arima_fit.summary())

# Make predictions
predictions = arima_fit.predict(start=len(data),
end=len(data)+10, dynamic=False)
print(predictions)
```

Here, we first import the necessary libraries and load the sunspot data. Then, we fit the ARIMA model with order (p, d, q) = (1, 1, 1) to the data. The summary() method of the fit model is then used to print model statistics, and the predict() method is used to make predictions on future values.

# BERT




| | |
|---|---|
| **Definition** | **Bidirectional Encoder Representations from Transformers (BERT) is a Transformer-based ML technique for natural language processing (NLP) pre-training developed by Google.** |
| **Main Domain** | **NLP & Speech Processing** |
| **Data Type** | **Text** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Not Explainable** |

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained transformer-based neural network model developed by Google. It's a state-of-the-art natural language processing (NLP) model that can be fine-tuned for various NLP tasks such as question answering, text classification, and named entity recognition.

One of the key features of BERT is its ability to understand the context of a word in a sentence using bidirectional training. Traditional language models, like word2vec or GloVe, are trained on the context in which a word appears, but only in one direction (either left or right). On the other hand, BERT is trained to understand a word's context in both directions. This allows BERT to understand the meaning of words in a sentence more accurately and to generate more accurate predictions.

BERT is pre-trained on a massive amount of text data, which allows it to be fine-tuned on smaller datasets for specific tasks. This makes BERT a versatile model that can be used in a wide range of NLP applications.

BERT has been very successful in NLP tasks and has set new state-of-the-art performance on several benchmark datasets, such as GLUE and SQuAD. It's also widely used in many industrial applications such as search engines, chatbots, and question-answering systems.

In summary, BERT is a pre-trained transformer-based neural network model that uses bidirectional training to understand the context of a word in a sentence; it's used in a wide range of NLP tasks and has set new state-of-the-art performance on several benchmark datasets.

Example:

Imagine we have a dataset of customer reviews for a product, and the task is to classify each review as positive or negative.

1. The first step is to fine-tune a pre-trained BERT model on the dataset. This is done by training the model on the labeled reviews and adjusting the model's parameters to the task at hand.
2. Once the model is fine-tuned, it can be used to classify new reviews. For example, given a recent review, "The product is great; it exceeded my expectations!" the model will classify it as positive.
3. BERT can also be used for more complex natural language understanding tasks, such as question answering or named entity recognition. For example, given a passage of text and a question, "What is the name of the company that developed BERT?", BERT can identify the answer "Google" by understanding the context of the question and the passage.

The example I provided is a simplified version of the process; in practice, the fine-tuning process can be more complex and require more data and computational resources. Also, BERT can solve a wide range of NLP tasks, not just classification.

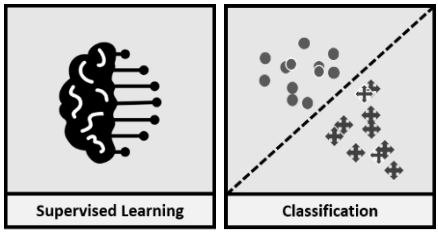
Code example of using the BERT (Bidirectional Encoder Representations from Transformers) model for fine-tuning a text classification task using the Hugging Face Transformers library in Python:

```
from transformers import BertForSequenceClassification,
AdamW, BertTokenizer
import torch

# Load the BERT model and the pre-trained weights
```

```
model =
BertForSequenceClassification.from_pretrained("bert-base-
uncased")

# Load the tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-
uncased")

# Prepare the input
text = "This is a sample sentence for classification."
encoded_input = tokenizer.encode_plus(text,
add_special_tokens=True, return_tensors="pt")
input_ids = encoded_input["input_ids"]
attention_masks = encoded_input["attention_mask"]

# Set up the optimizer
optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Train the model
model.train()
for _ in range(num_steps):
    optimizer.zero_grad()
    logits = model(input_ids, attention_masks)
    loss = criterion(logits[0], labels)
    loss.backward()
    optimizer.step()
```

Please note that this is a simple example, and in real-world tasks, you would need to handle tasks such as loading your data, preparing the data for training, training for multiple epochs, using a validation set, and so on.

# CONVOLUTIONAL NEURAL NETWORK




| | |
|---|---|
| **Definition** | **A convolutional neural network is a deep learning algorithm that takes an input, assigns meaning (learnable weights and biases) to different aspects/objects in the data, and can distinguish between them.** |
| **Main Domain** | **Computer Vision, NLP & Speech processing** |
| **Data Type** | **Image, video, text, time series** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Not Explainable** |

A convolutional neural network (CNN) is a type of deep learning neural network primarily used for image and video recognition and natural language processing tasks. CNNs are designed to process data that has a grid-like structure, such as an image composed of pixels arranged in a 2-D array.

The main building block of a CNN is the convolutional layer, which applies filters to the input data. These filters are small matrices that are used to extract features from the input data, such as edges, textures, and shapes. The filters are convolved with the input data, which means that they are multiplied element-wise with small regions of the input data, and then a sum is taken. This process is repeated for every location in the input data, resulting in a filtered output called a feature map.

The convolutional layers are followed by pooling layers, which are used to reduce the size of the feature maps. Pooling layers take the maximum or average value of a small region of the feature map and use it to represent that region. This helps reduce the data's dimensionality and make the model more robust to small changes in the input data.

Convolutional layers and pooling layers can be stacked to form a CNN, with multiple convolutional and pooling layers in sequence. After that, the network can have one or more fully connected layers that use the output of the convolutional layers to make the final prediction.

In summary, CNNs are neural networks that process grid-like data such as images, videos, and signals. They consist of multiple convolutional and pooling layers that extract features from the input data and one or more fully connected layers that make the final prediction. CNNs are widely used in computer vision, image and video analysis, and natural language processing tasks.

Example:

Imagine we have a dataset of images of dogs and cats, and the task is to train a model to classify each image as a dog or a cat.

1. The first step is to preprocess the images, which may include resizing, normalizing, and converting the images to arrays of pixels.
2. Next, we will create the CNN model, which may include multiple convolutional layers, pooling layers, and fully connected layers. The convolutional layers will apply filters to the input image to extract features such as edges, textures, and shapes. The pooling layers will reduce the size of the feature maps and make the model more robust to small changes in the input data.
3. The fully connected layers will take the output of the convolutional and pooling layers and use it to make the final prediction.
4. The model will be trained on the labeled images using a supervised learning algorithm, and the weights of the model will be adjusted to minimize the error between the predicted and actual labels.
5. Once the model is trained, it can be used to classify new images. For example, given a new image of a dog, the model will classify it as a dog.

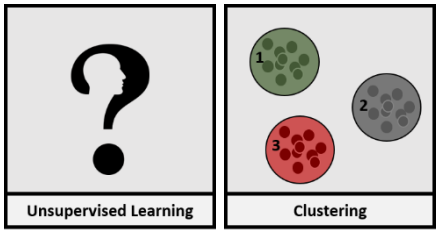
It's important to note that this is a simplified example; in practice, building a CNN model can be more complex and may require more data and computational resources. Also, CNNs can solve a wide range of image and video analysis tasks, not just classification.

Code example of a simple convolutional neural network implemented using the deep learning library Keras:

```
from keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense
from keras.models import Sequential

#create model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3,3),
activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

#compile model
model.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])
```

This example creates a simple convolutional neural network with two convolutional layers, each followed by a max pooling layer and two fully connected layers. This model is then compiled with the Adam optimizer and the categorical cross-entropy loss function and will be trained on a dataset with a shape of (28,28,1) image data and 10 output classes.

# DBSCAN




| | |
|---|---|
| **Definition** | **DBSCAN - Density-Based Spatial Clustering of Applications with Noise is a density-based, nonparametric clustering algorithm: given a set of points in a shared space, points that are close together (points with many near neighbors) are grouped and points that are alone in low-density regions (whose nearest neighbors are too far away) are marked as outliers.** |
| **Main Domain** | **Computer Vision** |
| **Data Type** | **Image** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups together points that are closely packed together (points with high density) and separates points that are sparsely located (points with low density).

DBSCAN defines a neighborhood around each point, called the "eps" neighborhood, and then groups all points within this neighborhood. Points that are in the same neighborhood are considered to be in the same cluster.

DBSCAN has two main parameters: eps and minPts. The eps parameter defines the radius of the neighborhood around each point, and the minPts parameter defines the minimum number of points required to form a dense region.

DBSCAN can find clusters of any shape and size, unlike other clustering algorithms like K-means, which assume that clusters are spherical and have similar sizes.

The algorithm proceeds in two steps:

1. For each point in the dataset, the algorithm retrieves all points in its eps-neighborhood.
2. It is considered a core point if a point has at least minPts points in its eps-neighborhood. All points in the eps-neighborhood of a core point belong to the same cluster.
3. Points not in the eps-neighborhood of any core point are considered noise points.

DBSCAN has several advantages over other clustering algorithms, such as not being required to specify the number of clusters in advance, and it can find clusters of any shape. However, it can be sensitive to the choice of the eps and minPts parameters, and it can be computationally expensive for large datasets.

In summary, DBSCAN is a density-based clustering algorithm that groups together points that are closely packed together and separates points that are sparsely located. It does not require to specify the number of clusters in advance and can find clusters of any shape, but it can be sensitive to the choice of the eps and minPts parameters and can be computationally expensive for large datasets.

Example:

Imagine we have a dataset of 2-dimensional points, and the task is to use DBSCAN to group the points into clusters.

1. We will start by defining the eps and minPts parameters. The eps parameter defines the radius of the neighborhood around each point, and the minPts parameter defines the minimum number of points required to form a dense region. For example, we can set eps = 1 and minPts = 5.
2. Next, we will run DBSCAN on the dataset. The algorithm will start by selecting an arbitrary point, and then retrieve all points in its eps-neighborhood. If the point has at least minPts points in its eps-neighborhood, it is considered a core point, and all points in its eps-neighborhood are considered to be in the same cluster.
3. The algorithm will then repeat the process for all other points in the dataset. Points that are not in the eps-neighborhood of any core point will be considered noise points.
4. After the algorithm is finished running, the points will be grouped into clusters based on their density. For example, the clusters might look like this:

5. Cluster 1: {(2, 3), (2, 4), (3, 4), (3, 5), (4, 5)} Cluster 2: {(10, 12), (11, 13), (12, 14)} Noise points: {(0, 0), (0, 1), (0, 2)}

It's important to note that this is a simplified example; in practice, the dataset can be much larger and more complex, and the eps and minPts parameters may need to be fine-tuned to get the best results.

Code example of DBSCAN in Python using the sklearn library:

```
from sklearn.cluster import DBSCAN
from sklearn import datasets

# load the iris dataset as an example
iris = datasets.load_iris()
X = iris.data

# create an instance of DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=10)

# fit the model to the dataset
dbscan.fit(X)

# retrieve the cluster labels
labels = dbscan.labels_

print(labels)
```

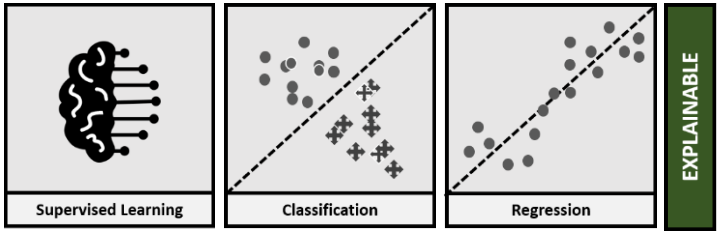
In this example, the DBSCAN algorithm is imported from the sklearn library and an instance of the algorithm is created with the following parameters:

- **eps**: the maximum distance between two samples for them to be considered as in the same neighborhood
- **min_samples**: the minimum number of samples in a neighborhood for a point to be considered as a core point

The fit method is then used to fit the model to the iris dataset, and the labels of each sample are retrieved using the labels_ attribute of the DBSCAN object.

Note that this is just an example, and the parameters of DBSCAN should be adjusted according to the specific dataset and problem you are working on.

# DECISION TREE




| | |
|---|---|
| **Definition** | **A decision tree is a diagram that uses a branching method to illustrate every possible output for a given input to decompose complex problems.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification, Regression** |
| **Explainability** | **Explainable** |

A decision tree is a supervised learning algorithm that is used for classification and regression tasks. It is a tree-based model, where each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label.

The decision tree algorithm starts with a root node representing the entire dataset. The algorithm then recursively splits the data into subsets based on the values of the input features. The goal of the splits is to create subsets of data that are as homogeneous as possible with respect to the target variable. The algorithm continues to split the subsets, creating new internal nodes, until a stopping criterion is met. The stopping criterion can be based on the number of instances in a leaf, the tree's depth, or the leaf's impurity.

At each internal node, the decision tree algorithm selects the feature and the threshold that results in the highest information gain. Information gain is a measure of how much the algorithm reduces the uncertainty of the target variable by doing a split. The feature and the threshold that result in the highest information gain are used to create the test at the node.

The decision tree algorithm can be used for both classification and regression tasks. In classification, the leaf nodes represent class labels, and the algorithm assigns the most common class label to a leaf node. In regression, the leaf nodes represent the mean or median value of the target variable for the instances in the leaf node.

Decision trees have several advantages: they are easy to understand and interpret, they can handle both categorical and numerical data, they are not sensitive to outliers, and they can be

visualized. However, they can be exposed to small changes in the training data and easily overfit it.

In summary, a Decision tree is a supervised learning algorithm used for classification and regression tasks; it's a tree-based model where each internal node represents a test on an attribute, each branch represents the outcome of the trial, and each leaf node represents a class label or a mean/median value of the target variable. Decision trees are easy to understand and interpret, but they can be sensitive to small changes in the training.

Example:

Imagine we have a dataset of customers, and the task is to use a decision tree to predict whether a customer will purchase a particular product. The dataset includes the customer's age, income, education level, and whether they have children.

1. The first step is to preprocess the data and split it into training and test sets.
2. Next, we will create the decision tree model using the training set. The algorithm starts with the root node, which represents the entire dataset. The algorithm then selects the feature and the threshold that result in the highest information gain and splits the dataset into subsets based on the values of the feature. The process is repeated recursively for each subset, creating new internal nodes until a stopping criterion is met.
3. After the decision tree is built, it can be used to make predictions on the test set. For example, given a new customer with the following attributes: age = 30, income = 50,000, education level = bachelor's, and children = yes,

the decision tree would predict that the customer will purchase the product.

4. The model can be evaluated using metrics such as accuracy, precision, recall, and f1-score.

It's important to note that this is a simplified example, and in practice, building a decision tree model can be more complex and require more data and computational resources. However, decision trees can also be used to solve many problems, including image and speech recognition, natural language processing, and many other types of issues.

Code example of decision tree implementation in Python using the scikit-learn library:

```
from sklearn import datasets
from sklearn import tree

# Load the iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Train a decision tree classifier
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)

# Predict the class for a new sample
sample = [[5, 4, 3, 2]]
prediction = clf.predict(sample)
print(prediction)
```

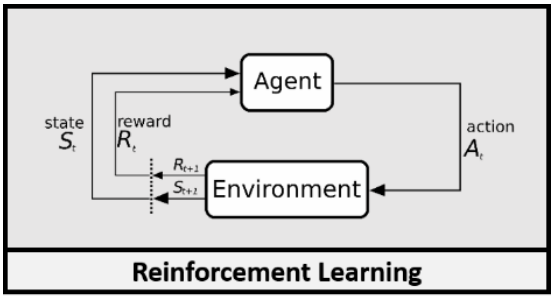
In this example, we first load the iris dataset, which includes 150 samples of iris flowers with four features (sepal length, sepal width, petal length, and petal width) and three classes (setosa, versicolor, and virginica).

Using the fit method, we then train a decision tree classifier on the data.

Finally, we use the classifier to predict the class for a new sample (in this case, a sample with sepal length 5, sepal width 4, petal length 3, and petal width 2) and print the predicted class.

Remember that this is just a simple example; in practice, decision trees can be more complex with many more features and different parameters.

# DEEP Q-LEARNING




| | |
|---|---|
| **Definition** | **Deep Q-learning works as a Q-learning algorithm because it uses a neural network to approximate the Q-value function to manage many states and actions.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Time Series** |
| **Data Environment** | **Reinforcement Learning** |
| **Learning Paradigm** | **Rewarding** |
| **Explainability** | **-** |

Deep Q-Learning is a variant of Q-Learning, an off-policy reinforcement learning algorithm that uses a deep neural network as a function approximator to approximate the Q-value function.

In Q-Learning, an agent learns to take action in an environment by trying to maximize the expected cumulative rewards over time. The agent uses a Q-value function, which assigns a value to each state-action pair to guide its decision-making process. The Q-value function represents the agent's estimate of the expected cumulative rewards if it starts in a given state, takes a given action, and follows a specific policy after that.

In traditional Q-learning, the Q-value function is represented as a table with one entry for each state-action pair. However, in many real-world problems, the state and action spaces can be vast, making it infeasible to represent the Q-value function as a table.

Deep Q-Learning addresses this problem by using a deep neural network as a function approximator to approximate the Q-value function. The neural network takes the state as input and outputs a prediction for the Q-value of each possible action. The network is trained to minimize the difference between the predicted Q-values and the true Q-values obtained by interacting with the environment.

One of the key features of deep Q-learning is that it uses a technique called experience replay, which allows the agent to learn from past experiences. The agent stores a replay buffer of past experiences, which consists of state-action-reward-next state tuples, and samples a random batch of experiences to learn from. This decorrelates the data and ensures that the agent learns from various experiences.

Deep Q-Learning has been successfully applied to many problems, including playing Atari games, controlling robots, and trading in

financial markets. It's a popular algorithm in the field of reinforcement learning and has led to many advances.

In summary, Deep Q-Learning is a variant of Q-Learning algorithm that uses a deep neural network as a function approximator to approximate the Q-value function; it allows learning in a high dimensional space, it uses experience replay technique to learn from past experiences, and it's a popular algorithm in the field of reinforcement learning.

Example:

Imagine we have an agent that controls a robotic arm, and the task is to train the agent to reach and grab a target object in a 3D environment.

1. The first step is to define the agent's state space, action space, and reward function. The state space might include the position and orientation of the robotic arm, the position of the target object, and other relevant information, such as the distance between the component and the object. The action space might include the arm's actions, such as moving left, right, up, or down. The reward function would assign a high reward for successfully grasping the object and a low reward for any other outcome.
2. Next, we will create the deep Q-network, a neural network that takes the state as input and outputs the Q-value of each possible action. The network is trained using the Q-learning algorithm to minimize the difference between the predicted Q-values and the true Q-values, which are obtained by interacting with the environment.

3. The agent interacts with the environment by taking action and observing the resulting state and reward. It stores this information in a replay buffer and samples a random batch of past experiences to learn from.
4. After the agent has interacted with the environment for sufficient time, it will have learned an optimal policy for reaching and grasping the target object.
5. Once the agent has learned the optimal policy, it can use it to reach and grab the object in new situations.

It's important to note that this is a simplified example. In practice, the process of training a deep Q-learning agent can be more complex and may require more data and computational resources; also the robotic arm example I provided is just one of the many examples of how deep Q-learning can be used; it could be applied to different problems such as playing video games, financial trading and many other problems where decision making is needed.

An example of code for implementing a deep Q-learning algorithm using Python and the Keras library:

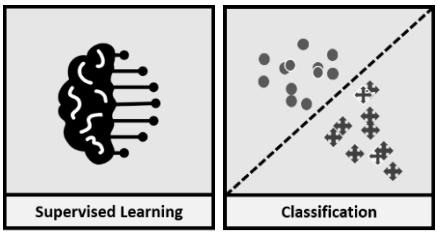
```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

# create the Q-network
model = Sequential()
model.add(Dense(24, input_shape=(state_size,),
activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(action_size, activation='linear'))
model.compile(loss='mse',
optimizer=Adam(lr=learning_rate))

# training loop
for episode in range(episodes):
    state = env.reset()
```

```
    state = np.reshape(state, [1, state_size])
    for time in range(max_steps):
        # choose an action
        action = np.argmax(model.predict(state))
        # take the action and observe the next state and
reward
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1,
state_size])
        # update the Q-network
        target = reward + discount_factor *
np.amax(model.predict(next_state))
        target_vec = model.predict(state)
        target_vec[0][action] = target
        model.fit(state, target_vec, epochs=1, verbose=0)
        state = next_state
        if done:
            break
```

This code uses the Q-learning algorithm to train a neural network to play a simple game or control a simulated agent. The Q-network takes the current state of the environment as input and outputs a prediction of the expected future reward for each possible action. The algorithm uses these predictions to select the action that maximizes the expected reward and then updates the Q-network based on the observed reward and the next state of the environment. The example uses the Keras library to define and train the Q-network.

Please note this is an example and may need some dependencies to run it on your machine.

# EFFICIENTNET




| | |
|---|---|
| **Definition** | **EfficientNet is a Convolutional Neural Network based on depth-wise convolutions, making it lighter than other CNNs. It also allows scaling the model with a unique lever: the compound coefficient.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Image** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Not Explainable** |

EfficientNet is a family of convolutional neural networks (CNNs) developed by Google Research that aims to improve the accuracy and efficiency of CNNs. The EfficientNet models are designed to be scalable in terms of their architecture, input resolution, and computational resources.

The main idea behind EfficientNet is to scale up the architecture, input resolution, and computational resources of CNNs in a principled and consistent way. Therefore, the models use a compound scaling method, which adjusts the network's depth, width, and resolution in a multi-dimensional way. This allows the models to achieve a good balance between accuracy and efficiency.

EfficientNet models are built using the AutoML Mobile framework, which uses a neural architecture search (NAS) algorithm to find the best architecture for a given computational budget. The NAS algorithm searches for the best combination of depth, width, and resolution and other architectural elements such as skip connections and pooling operations.

The EfficientNet models have achieved state-of-the-art performance on various image classification benchmarks, such as ImageNet and COCO, while using fewer computational resources than other architectures.

EfficientNet models are also lightweight, which makes them suitable for deployment on mobile and embedded devices with limited computational resources.

In summary, EfficientNet is a family of convolutional neural networks developed by Google Research that aims to improve the accuracy and efficiency of CNNs. They are designed to be scalable in terms of architecture, input resolution, and computational

resources. EfficientNet models have achieved state-of-the-art performance on various image classification benchmarks and are lightweight, making them suitable for deployment on mobile and embedded devices.

Example:

Imagine we have a dataset of images of animals, and the task is to use an EfficientNet model to classify the images into different animal classes.

1. The first step is to preprocess the data and split it into training and test sets.
2. Next, we will create an EfficientNet model using the training set. The model can be built using a pre-trained EfficientNet model or the AutoML Mobile framework to find the best architecture for a given computational budget.
3. After the model is trained, it can be used to make predictions on the test set. For example, given an image of a dog, the EfficientNet model would predict that the image belongs to the "dog" class with a high probability.
4. The model's performance can be evaluated using metrics such as accuracy, precision, recall, and f1-score.
5. Once the model is fine-tuned, and its performance is satisfactory, it can be deployed to make predictions on new images.

It's important to note that this is a simplified example, and in practice, building and fine-tuning an EfficientNet model can be more complex and require more data and computational resources. EfficientNet models are suitable for a wide range of

computer vision problems, including object detection, semantic segmentation, and many other types of issues.

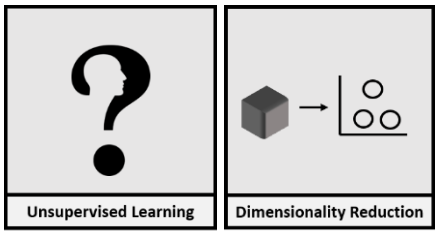
An example of how to use the EfficientNetB0 model in a Keras workflow:

```
from efficientnet import EfficientNetB0

model = EfficientNetB0(weights='imagenet')
```

This will load the EfficientNetB0 model with pre-trained weights on the ImageNet dataset. Then, you can use this model for image classification tasks by calling **model.predict(data)** on your input data, where **data** is a 4D numpy array with shape (batch_size, height, width, channels).

Please note that this is a very simple example, in practice you will need to add additional layers and code to handle loading and preprocessing of data, training, evaluation and so on.

# FACTOR ANALYSIS OF CORRESPONDENCES




| | |
|---|---|
| **Definition** | **The factorial correspondence analysis (CFA) is a statistical method of data analysis that allows the analysis and prioritization of the information contained in a rectangular table of data and which is mainly used to study the link between two variables (qualitative or categorical).** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Dimension Reduction** |
| **Explainability** | |

Factor analysis of correspondences (FAC) is a statistical technique used to analyze categorical data. It is a variation of factor analysis, a technique for identifying the underlying structure of a set of variables.

In FAC, the data consists of a set of categorical variables and observations, where each observation is a set of responses to the categorical variables. FAC aims to identify a smaller set of latent factors that explain the relationships among the categorical variables.

FAC starts by creating a contingency table, which shows the frequencies of the responses to the categorical variables. The contingency table is then transformed into a matrix of dissimilarities, which measures the dissimilarity between the responses to the categorical variables. The dissimilarity matrix is then used as input to a factor analysis algorithm, such as principal component analysis (PCA), to identify the latent factors.

The number of factors that are retained is based on the scree plot or eigenvalue. The elements are the variables' linear combinations, and the factor scores are the linear combinations of the observations.

FAC can be used to identify patterns in the relationships among the categorical variables, and it can be used to reduce the dimensionality of the data. It's used to find underlying structures in the data; for example, in survey data, where multiple questions are used to gather information about different aspects of a concept, FAC can be used to identify the underlying factors that are related to the concept.

In summary, Factor Analysis of Correspondences (FAC) is a statistical technique that is used to analyze categorical data; it's a

variation of factor analysis, and the goal of FAC is to identify a smaller set of latent factors that explain the relationships among the categorical variables. FAC starts by creating a contingency table, which shows the frequencies of the responses to the categorical variables, it's used to find underlying structure in the data, and it can be used to reduce the dimensionality of the data.

Example:

Imagine we have a customer survey dataset, and the task is to use FAC to identify the underlying factors that influence customer satisfaction. The survey includes questions about different aspects of customer service, such as wait time, service quality, and staff friendliness.

1. The first step is to preprocess the data and create a contingency table showing the responses to the survey questions.
2. Next, we will create a matrix of dissimilarities, which measures the dissimilarity between the responses to the survey questions. The dissimilarity matrix is then used as input to a factor analysis algorithm, such as principal component analysis (PCA), to identify the latent factors.
3. The number of factors to retain is based on the scree plot or eigenvalue, The elements are the linear combinations of the variables, and the factor scores are the linear combinations of the observations.
4. After the factors are identified, we can use them to interpret the relationships among the survey questions. For example, the first factor might represent "wait time," the second factor might represent "quality of service,"

and the third factor might represent "friendliness of the staff."

5. Once the factors are identified, we can use them to analyze the data and identify patterns in the relationships among the survey questions. For example, customers dissatisfied with the wait time are also dissatisfied with the quality of service.

It's important to note that this is a simplified example, and in practice, using FAC to analyze survey data can be more complex and may require more data and computational resources. Nevertheless, FAC can be used in various applications, including market research, social science research, and many other types of problems where categorical data is analyzed.

# GAN


| | |
|---|---|
| **Definition** | **A GAN is a generative model where two networks are placed in the competition. The first model is the generator; it generates a sample (e.g., an image), while its opponent, the discriminator, tries to detect whether a sample is real or the result of the generator. Both improve on the performance of the other.** |
| **Main Domain** | **Computer Vision** |
| **Data Type** | **Image, Video** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **-** |
| **Explainability** | **-** |

Generative Adversarial Networks (GANs) are a type of deep learning architecture for generative models. A GAN consists of two neural networks, a generator, and a discriminator, that are trained together in a game-theoretic manner.

The generator is a neural network that learns to generate new data samples similar to the training data. The generator takes a random noise vector as input and produces a synthetic data sample as output. The generator's goal is to generate samples indistinguishable from the training data.

The discriminator is a neural network that learns to distinguish between the synthetic data samples generated by the generator and the actual training data. The discriminator takes a data sample as input and produces a scalar value representing the probability that the input sample is real. The goal of the discriminator is to identify the synthetic samples correctly.

The generator and the discriminator are trained together in a game-theoretic manner, where the generator tries to produce samples that can fool the discriminator, and the discriminator tries to identify the synthetic samples correctly. The training continues until the generator has indistinguishable samples from the real data.

GANs have been used for various tasks such as image generation, text-to-speech, video generation, and many other problems. They can generate high-quality synthetic data samples that are indistinguishable from real data.

In summary, GAN stands for Generative Adversarial Networks, a type of deep learning architecture for generative models. It consists of two neural networks, a generator, and a discriminator, that are trained together in a game-theoretic manner. The

generator is a neural network that learns to generate new data samples similar to the training data, and the discriminator is a neural network that learns to distinguish between the synthetic data samples generated by the generator and the actual training data. Although, GANs have been used for a variety of tasks such as image generation, text-to-speech, video generation, and many other types of problems; they have been shown to be able to generate high-quality synthetic data samples that are indistinguishable from real data.

Example:

Imagine we have a dataset of faces, and the task is to use a GAN to generate new images of faces that are similar to the training data.

1. The first step is to preprocess the data and split it into training and test sets.
2. Next, we will define the generator and discriminator neural networks. The generator takes a random noise vector as input and produces an image as output. The discriminator takes an image as input and produces a scalar value representing the probability that the input image is real.
3. We will then train the generator and discriminator together in a game-theoretic manner. The generator will try to produce images that can fool the discriminator, while the discriminator will try to identify the synthetic images correctly. The training process continues until the generator produces indistinguishable images from the real data.

4. After the GAN is trained, it can be used to generate new images of faces. For example, given a random noise vector, the GAN would generate a new image of a face that is similar to the faces in the training data.
5. The generated images can be evaluated by a human evaluator or by using a pre-trained classifier to check how similar the generated images are to the real images.

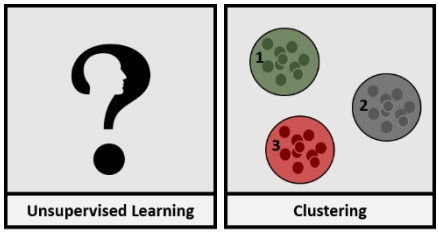
It's important to note that this is a simplified example, and in practice, the process of building and training a GAN can be more complex and may require more data and computational resources. GANs have been used for a variety of tasks such as image generation, text-to-speech, video generation, and many other types of problems.

Code example of GAN (Generative Adversarial Network) implemented in Python using the TensorFlow library:

```
import tensorflow as tf

# Define the generator model
def generator_model():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Dense(7*7*256,
use_bias=False, input_shape=(100,)))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU())

    model.add(tf.keras.layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)

    model.add(tf.keras.layers.Conv2DTranspose(128, (5,
5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU())

    model.add(tf.keras.layers.Conv2DTranspose(64, (5, 5),
strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
```

```
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU())

    model.add(tf.keras.layers.Conv2DTranspose(1, (5, 5),
strides=(2, 2), padding='same', use_bias=False,
activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model

# Define the discriminator model
def discriminator_model():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(64, (5, 5),
strides=(2, 2), padding='same',
                                     input_shape=[28, 28,
1]))
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Dropout(0.3))

    model.add(tf.keras.layers.Conv2D(128, (5, 5),
strides=(2, 2), padding='same'))
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Drop
```

# GMM




| | |
|---|---|
| **Definition** | **A Gaussian mixture model is a probabilistic model that assumes that all data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.** |
| **Main Domain** | **Computer Vision, NLP & Speech Processing** |
| **Data Type** | **Text, Time Series, Image, Video,** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

A Gaussian Mixture Model (GMM) is a statistical model used to represent a set of data as a mixture of multiple Gaussian distributions. GMMs are a type of probability density function that can be used for unsupervised learning tasks such as clustering and density estimation.

A GMM is defined by a set of parameters, including the Gaussian distributions' means, variances, and mixing coefficients. The mixing coefficients represent the proportion of the data that belongs to each Gaussian distribution.

The GMM algorithm starts by initializing the parameters of the Gaussian distributions. The algorithm then uses the Expectation-Maximization (EM) algorithm to iteratively update the parameters until they converge to the maximum likelihood estimates.

1. The EM algorithm is a two-step process:
2. The E-step: computes the probability that each data point belongs to each Gaussian distribution

The M-step: updates the parameters of the Gaussian distributions based on the probabilities calculated in the E-step

The process is repeated until the parameters converge to a stable solution.

GMM can be used for various unsupervised learning tasks, such as density estimation, clustering, and anomaly detection. It's also used as a generative model, where it can be used to generate new samples from the data distribution.

In summary, Gaussian Mixture Model (GMM) is a statistical model used to represent a data set as a mixture of multiple Gaussian distributions. It's a probability density function that can be used for unsupervised learning tasks such as clustering and density

estimation. GMM is defined by a set of parameters, including the Gaussian distributions' means, variances, and mixing coefficients. The GMM algorithm starts by initializing the parameters of the Gaussian distributions and uses the Expectation-Maximization (EM) algorithm to iteratively update the parameters until they converge to the maximum likelihood estimates. GMM can be used for various unsupervised learning tasks, such as density estimation, clustering, anomaly detection, and as a generative model.

Example:

Imagine we have a dataset of two-dimensional points that represent the locations of customers in a shopping mall. The task is to use GMM to cluster the customers into different groups based on their location.

1. The first step is to preprocess the data and plot the points on a scatter plot to visualize the distribution of the data.
2. Next, we will initialize the parameters of the Gaussian distributions, such as the means and variances, based on the data distribution. This can be done by randomly selecting k points from the data and using them as the initial means of the k Gaussian distributions.
3. We will then use the Expectation-Maximization (EM) algorithm to iteratively update the parameters of the Gaussian distributions until they converge to the maximum likelihood estimates.
4. After the GMM is trained, it can be used to assign each data point to one of the k clusters based on the probability that it belongs to each Gaussian distribution.

5. The clusters can be visualized by plotting the data points and coloring them according to the cluster they belong to.

It's important to note that this is a simplified example, and in practice, the process of building and training a GMM can be more complex and may require more data and computational resources. The GMM model is sensitive to the initial values, so it's essential to choose the initial values carefully. GMM can be used for a variety of unsupervised learning tasks, such as density estimation, clustering, anomaly detection, and as a generative model.

Code example of how to fit a Gaussian mixture model (GMM) to a dataset using the scikit-learn library in Python:

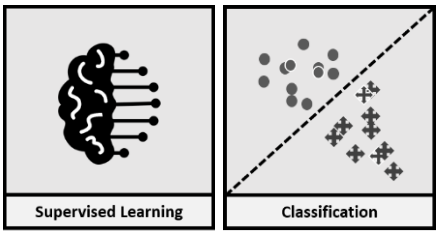
```
from sklearn.mixture import GaussianMixture
import numpy as np

# Generate some sample data
np.random.seed(0)
data = np.random.randn(100, 2)

# Fit a GMM to the data
gmm = GaussianMixture(n_components=3)
gmm.fit(data)

# Predict the cluster assignments for each data point
labels = gmm.predict(data)

# Print the cluster means and covariances
print("Cluster Means:", gmm.means_)
print("Cluster Covariances:", gmm.covariances_)
```

In this example, we first generate sample data using the numpy library. The data is a 2D array of 100 rows and 2 columns. Then, we create an instance of the GaussianMixture class with 3 components and fit it to the data using the **fit** method. After

fitting, we use the **predict** method to predict the cluster assignments for each data point. Finally, we print the cluster means and covariances using the **means_** and **covariances_** attributes of the GMM object, respectively.

Please note that this is just one example of how GMM can be used, and the specific code will depend on the dataset and the problem you are trying to solve.

# GPT-3




| | |
|---|---|
| **Definition** | **Generative Pre-trained Transformer 3 (GPT-3) is an autoregressive language model that uses Deep Learning to generate human-like text.** |
| **Main Domain** | **NLP & Speech Processing** |
| **Data Type** | **Text** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Not Explainable** |

GPT-3 (Generative Pre-trained Transformer 3) is a state-of-the-art language generation model developed by OpenAI. It is one of the largest and most powerful language models to date, with over 175 billion parameters. GPT-3 is pre-trained on a massive amount of text data, and it can be fine-tuned to perform a wide range of natural language processing tasks, such as language translation, text summarization, text completion, and question answering.

GPT-3 is based on the transformer architecture, which is a type of neural network that is designed to process sequential data, such as text. The transformer architecture uses self-attention mechanisms, which allow the model to weigh the importance of different parts of the input when making predictions.

One of the key features of GPT-3 is its ability to generate human-like text. It can complete a given prompt or question with coherent and fluent text. It can also generate new text, such as writing a story or a news article, with a level of quality that is often hard to distinguish from text written by humans.

GPT-3 has been used in a variety of applications, from content generation to chatbots to language translation. Due to its vast knowledge and capability of understanding context GPT-3 is being used for multiple tasks like question answering, summarization, and text completion, among others.

In summary, GPT-3 (Generative Pre-trained Transformer 3) is a state-of-the-art language generation model developed by OpenAI. It's one of the largest and most powerful language models to date, with over 175 billion parameters; it's pre-trained on a massive amount of text data, and it can be fine-tuned to perform a wide range of natural language processing tasks, such as language translation, text summarization, text completion, and question answering, It's based on the transformer architecture, and it has

the ability to generate human-like text, GPT-3 has been used in a variety of applications, from content generation to chatbots to language translation.

Example:

Imagine we want to build a chatbot to answer customer questions about a product. We can use GPT-3 to fine-tune the model on the customer questions and answers dataset.

1. The first step is to preprocess the data by creating a dataset of customer questions and their corresponding answers.
2. Next, we fine-tune GPT-3 on this dataset by training it to predict the correct answer given a customer question.
3. After training, the model can be deployed as a chatbot that takes customer questions as input and generates an answer based on the training data.
4. The chatbot can be integrated with a customer service platform, such as a website or a mobile application, to provide real-time customer support.
5. The chatbot can also be tested by asking new questions and evaluating its performance.

It's important to note that this is a simplified example, and in practice, fine-tuning GPT-3 and deploying a chatbot can be more complex and may require more data and computational resources. GPT-3 has been used for a wide range of tasks, such as language translation, text summarization, text completion, and question answering. With its vast knowledge and ability to understand context, GPT-3 has the potential to be used in a wide range of applications in the future.

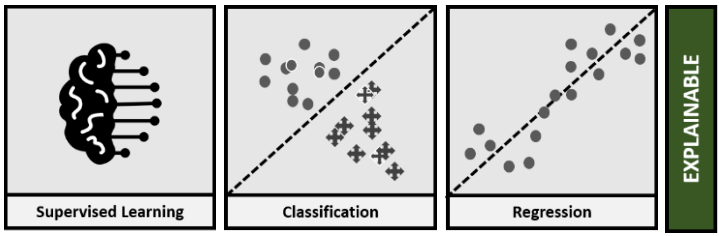
To use GPT-3 API, you will first need to sign up for an API key from OpenAI. Once you have the API key, you can then use it to make requests to the GPT-3 API. The API allows you to input a prompt, and it will generate a response based on that prompt.

```
import openai_secret_manager

# Get API key
secrets = openai_secret_manager.get_secrets("openai")
api_key = secrets["api_key"]

# Install the OpenAI library
!pip install openai

# Import the library
import openai

# Set the API key
openai.api_key = api_key

# Define the prompt
prompt = "What is the capital of France?"

# Make the API call
response = openai.Completion.create(engine="text-davinci-
002", prompt=prompt)

# Print the response
print(response["choices"][0]["text"])
```

The above code snippet is using the openai python library which provides the methods to interact with the GPT3 API. The **openai.Completion.create()** method is used to create a completion of the prompt given in the prompt variable. The **response["choices"][0]["text"]** will give the text response generated by the model.

Please keep in mind that GPT-3 is a paid service and you may incur charges depending on your usage.

# GRADIENT BOOSTING MACHINE




| | |
|---|---|
| **Definition** | **Gradient boosting is a technique that optimizes a decision tree by combining weak models to improve model prediction.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification, Regression** |
| **Explainability** | **Explainable** |

Gradient Boosting Machine (GBM) is an ensemble machine learning algorithm that combines the predictions of multiple weaker models to create a stronger model. GBM is a boosting algorithm that trains weak models sequentially, where each model tries to correct the mistakes of the previous model.

GBM is a robust algorithm for both classification and regression problems. It works by iteratively training weak models, such as decision trees, and then combining their predictions to create a stronger model. The algorithm starts with a simple model and then improves it by adding more complex models that correct the mistakes of the previous models.

The main idea behind GBM is to use the gradient descent algorithm to optimize the parameters of the weak models. This is done by computing the gradient of the loss function concerning the parameters of the model and updating the parameters in the direction that minimizes the loss.

GBM algorithm can be used with different weak models, such as decision trees, linear regression, and neural networks, among others. It can also be used with different loss functions, such as mean squared error for regression problems and cross-entropy for classification problems.

In summary, Gradient Boosting Machine (GBM) is an ensemble machine learning algorithm that combines the predictions of multiple weaker models to create a stronger model. GBM is a boosting algorithm that trains weak models sequentially, where each model tries to correct the mistakes of the previous model. GBM is a powerful algorithm for both classification and regression problems. It works by iteratively training weak models and then combining their predictions to create a stronger model. GBM algorithm can be used with different types of weak models, such

as decision trees, linear regression, and neural networks, and it can be used with different loss functions, such as mean squared error for regression problems and cross-entropy for classification problems.

Example:

Imagine we have a dataset of customer information, such as age, income, and education level, and we want to use GBM to predict whether a customer will default on a loan.

1. The first step is to preprocess the data and split it into training and test sets.
2. Next, we will initialize the GBM with a simple model, such as a decision tree with a single split.
3. We will then use the gradient descent algorithm to optimize the parameters of the decision tree. This is done by computing the gradient of the loss function concerning the parameters of the model and updating the parameters in the direction that minimizes the loss.
4. After the first decision tree is trained, we will add another decision tree to the ensemble and train it to correct the mistakes of the previous tree.
5. We will repeat the process of adding decision trees and training them until the ensemble's performance on the test set stops improving.
6. After the GBM is trained, it can be used to predict whether a customer will default on a loan by combining the predictions of all the decision trees in the ensemble.

It's important to note that this is a simplified example, and in practice, building and training a GBM can be more complex and require more data and computational resources. GBM algorithm

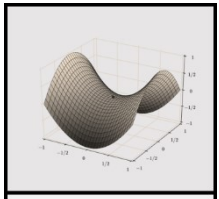
can be used with different types of weak models and different loss functions, so it's essential to choose the right type of weak model and loss function for the specific problem. Nevertheless, GBM is a powerful algorithm, and it has been used in various applications, such as credit risk assessment, customer churn prediction, and fraud detection, among others.

An example of using gradient boosting in Python with the popular library XGBoost:

```
import xgboost as xgb

# load data
data = ...

# split data into training and testing sets
X_train, X_test, y_train, y_test = ...

# create a gradient boosting model
gbm = xgb.XGBClassifier(
    max_depth=3,
    n_estimators=300,
    learning_rate=0.05
)

# train the model
gbm.fit(X_train, y_train)

# make predictions on the test set
y_pred = gbm.predict(X_test)
```

Please note that this is just a basic example, and the parameters used here (such as the maximum depth of the trees, the number of estimators, and the learning rate) may need to be adjusted for your specific dataset.

# GRADIENT DESCENT



Optimization

| | |
|---|---|
| **Definition** | **Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the function's gradient (or approximate gradient) at the current point because this is the direction of steepest descent.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **-** |
| **Learning Paradigm** | **Optimization** |
| **Explainability** | **-** |

Gradient descent is an optimization algorithm used to minimize a function, typically a cost function, by iteratively moving in the direction of the steepest decrease of the function. It is a first-order optimization algorithm which uses the function's gradient to determine the direction of the next step. The algorithm aims to find the minimum of the function, which corresponds to the set of parameters that result in the lowest cost.

The basic idea behind gradient descent is to start with an initial set of parameters for a model and then iteratively update the parameters in the direction of the negative gradient of the cost function. The gradient is the vector of the partial derivatives of the cost function concerning the parameters, and it points in the direction of the steepest cost function increase.

The algorithm proceeds iteratively by computing the gradient at the current parameter values and then taking a step in the direction opposite to the gradient to reduce the cost. The step size is determined by a learning rate parameter, which controls how large the step is in the opposite direction of the gradient.

Different variations of gradient descent algorithms include Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and the famous Adaptive Gradient Algorithm (Adam).

In summary, Gradient descent is an optimization algorithm used to minimize a function by iteratively moving in the direction of the steepest decrease of the function; it's a first-order optimization algorithm, which means that it uses the gradient of the function to determine the direction of the next step. The algorithm aims to find the minimum of the function, which corresponds to the set of parameters that result in the lowest cost. The basic idea behind gradient descent is to start with an initial set of parameters for a model and then iteratively update the parameters in the direction

of the negative gradient of the cost function. Different variations of gradient descent algorithms include Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and the popular Adaptive Gradient Algorithm (Adam).

Example:

Imagine a simple linear regression model with one input feature and one output. The goal is to find the best set of parameters that minimize the mean squared error (MSE) between the predicted and actual output values.

1. The first step is to initialize the model parameters with random values.
2. Next, we will calculate the gradient of the MSE concerning the parameters. The gradient is the vector of partial derivatives of the MSE concerning the parameters.
3. We will then update the parameters by subtracting the gradient multiplied by a learning rate from the current parameter values.
4. We will repeat the process of calculating the gradient and updating the parameters until the parameters converge to a stable solution.
5. After the model is trained, it can be used to make predictions on new input data.

It's important to note that this is a simplified example, and in practice, training a model using gradient descent can be more complex and may require more data and computational resources. Additionally, the cost function can be non-convex, so the algorithm may get stuck in a local minimum instead of finding the global minimum. To mitigate this, there are variations of the

Gradient Descent algorithm, such as Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and the popular Adaptive Gradient Algorithm (Adam), which have been developed to overcome these issues.

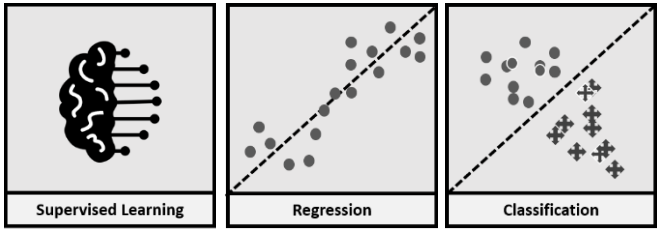
An example of gradient descent implemented in Python using the popular machine learning library scikit-learn:

In this example, the model is a linear regression model implemented using the SGDRegressor class. The loss parameter is set to 'squared_loss' to minimize the mean squared error, and the penalty parameter is set to None to not use any regularization. The max_iter parameter is set to 1000, and the learning_rate is set to 'constant' with a value of 0.1. The fit method is used to train the model on the X_train and y_train data, and the predict method is used to make predictions on the X_test data.

```
from sklearn.linear_model import SGDRegressor

# create an instance of the model
sgd = SGDRegressor(loss='squared_loss', penalty=None,
max_iter=1000, learning_rate='constant', eta0=0.1)

# fit the model to the training data
sgd.fit(X_train, y_train)

# make predictions on the test data
y_pred = sgd.predict(X_test)
```

This is a simple example; in the real world, you may want to add more features, improve the model performance and optimize the learning rate.

# GRAPH NEURAL NETWORKS




| | |
|---|---|
| **Definition** | **Graph neural networks (GNNs) are deep learning-based methods that operate on graph domains. Graphs are a kind of data structure that models a set of objects (nodes) and their relationships (edges)** |
| **Main Domain** | **Computer Vision, Speech Processing** |
| **Data Type** | **Image** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Regression, Classification** |
| **Explainability** | **-** |

Graph Neural Networks (GNNs) is a type of neural network that can process graph-structured data. GNNs are designed to handle non-Euclidean structured data, unlike traditional neural networks that are designed to handle vectorial data. Graph-structured data includes entities and their relationships, such as social networks, molecule structures, road networks, and many more.

GNNs are based on the idea of message passing, where each node in the graph sends and receives information from its neighboring nodes in the form of feature vectors. These feature vectors are then used to update the representation of each node. The updated representations are then used to make predictions, such as node classifications, link predictions, and graph classification.

There are different types of GNNs, such as Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Graph Autoencoders. Every kind of GNN uses a different method to process graph-structured data and make predictions.

In summary, Graph Neural Networks (GNNs) is a type of neural network that can process graph-structured data; they are designed to handle non-Euclidean structured data, unlike traditional neural networks designed to handle vectorial data. GNNs are based on the idea of message passing, where each node in the graph sends and receives information from its neighboring nodes in the form of feature vectors. GNNs can be used for node classification, link prediction, and graph classification. There are different types of GNNs, such as Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Graph Autoencoders.

Example:

Imagine we have a social network dataset with users and their connections, and we want to use a GNN to predict the likelihood of a user forming a new relationship with another user.

1. The first step is to preprocess the data and convert it into a graph structure, where each user is represented as a node and each connection is represented as an edge between nodes.
2. Next, we will initialize the GNN with an initial representation of the nodes, such as a one-hot encoding of the node's ID.
3. We will then use a message-passing mechanism, such as graph convolution, to update the representation of each node by aggregating information from its neighboring nodes.
4. After several layers of message passing, we will use the updated representations of the nodes to make predictions, such as the likelihood of a new connection being formed between two nodes.
5. The GNN can be trained by providing labeled connections and adjusting the model's parameters to minimize the error between the predicted and actual connections.
6. After the GNN is trained, it can be used to make predictions on new connections by using the updated representations of the nodes.

It's important to note that this is a simplified example, and in practice, training a GNN can be more complex and may require more data and computational resources. Additionally, there are different types of GNNs, such as Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Graph Autoencoders, each with their way of processing and making predictions on graph-structured data, so it's essential to choose the right type of GNN for the specific problem. GNNs have been

used in various applications, such as social network analysis, recommendation systems, and drug discovery, among others.

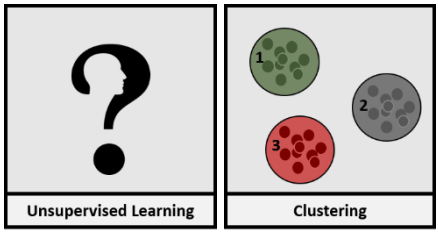
An example code for a simple graph neural network (GNN) implemented in PyTorch; This code defines a basic GNN model and applies it to a graph classification task on the Cora dataset.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class GNN(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(GNN, self).__init__()
        self.conv1 = nn.Linear(input_dim, hidden_dim)
        self.conv2 = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, x, adj):
        x = F.relu(self.conv1(x))
        x = F.dropout(x, training=self.training)
        x = self.conv2(x)
        x = F.dropout(x, training=self.training)
        x = torch.mm(adj, x)
        return x

class GNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim,
output_dim):
        super(GNNModel, self).__init__()
        self.gnn = GNN(input_dim, hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x, adj):
        x = self.gnn(x, adj)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)

# create the model
model = GNNModel(input_dim=1433, hidden_dim=16,
output_dim=7)

# define the loss function and optimizer
criterion = nn.NLLLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

```
# training loop
for epoch in range(200):
    # forward pass
    output = model(x, adj)
    loss = criterion(output, labels)

    # backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

This code is an example, and it might not run as it is, but it should give an idea of how a GNN can be implemented in PyTorch.

# HIERARCHICAL CLUSTERING




| | |
|---|---|
| **Definition** | **Hierarchical clustering is a cluster analysis method that seeks to build a hierarchy of clusters. The result is a tree-based representation of the objects, named a dendrogram.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

Hierarchical Clustering is a method of clustering that creates a hierarchical representation of the clusters. It is a type of clustering algorithm that builds a tree-like structure, called a dendrogram, to represent the clusters. The dendrogram shows the relationships between the clusters, and how the clusters are merged to form larger clusters.

There are two main types of hierarchical clustering: Agglomerative and Divisive.

Agglomerative hierarchical clustering starts with each data point as its own cluster and then iteratively merges the closest clusters until all the data points are in a single cluster.

Divisive hierarchical clustering, on the other hand, starts with all data points in a single cluster, then iteratively splits the cluster into smaller clusters until each data point is in its own cluster.

Hierarchical Clustering has some advantages over other clustering methods, such as K-Means Clustering. It is more flexible and doesn't require setting the number of clusters in advance. Also, it can be used to identify natural groupings in the data and allow for the representation of intermediate clusters, which can be useful for exploratory data analysis. However, it can be computationally expensive for large datasets.

In summary, Hierarchical Clustering is a method of clustering that creates a hierarchical representation of the clusters, it is a type of clustering algorithm that builds a tree-like structure, called a dendrogram, to represent the clusters. There are two main types of hierarchical clustering: Agglomerative and Divisive. Agglomerative hierarchical clustering starts with each data point as its own cluster and then iteratively merges the closest clusters until all the data points are in a single cluster. On the other hand, divisive hierarchical clustering starts with all data points in a single

cluster and then iteratively splits the cluster into smaller clusters until each data point is in its own cluster. Hierarchical Clustering has some advantages over other clustering methods, such as K-Means Clustering. It is more flexible and doesn't require setting the number of clusters in advance. However, it can be computationally expensive for large datasets.

Example:

Imagine we have a dataset of 10 customers with their purchasing history, and we want to use hierarchical clustering to group the customers into clusters based on their purchasing habits.

1. The first step is preprocessing the data and computing the pairwise distances between customers. A typical distance metric used in hierarchical clustering is Euclidean distance.
2. Next, we will start with each customer as its own cluster and create a dendrogram to represent the relationships between the clusters.
3. We will then iteratively merge the closest clusters according to their distance.
4. We will continue merging the closest clusters until all the customers are in a single cluster.
5. At this point, we can use a threshold distance to cut the dendrogram and identify the clusters.
6. After the clusters are identified, we can analyze each cluster's characteristics to understand the customers' purchasing habits.

It's important to note that this is a simplified example, and in practice, the process of performing hierarchical clustering can be

more complex and may require more data and computational resources. Hierarchical clustering can be performed using different linkage methods, such as single, complete, and average, each of which defines the distance between clusters differently. The linkage method affects the shape of the resulting dendrogram and the clusters formed.

An example of hierarchical clustering using the Scikit-learn library in Python:

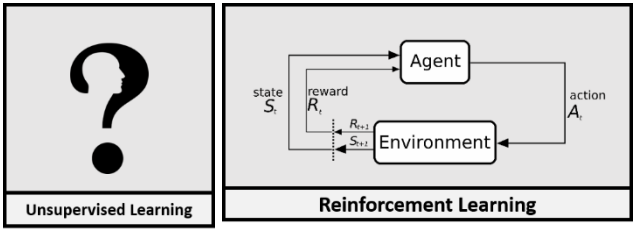
```
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt

# Generate sample data
X, y = make_blobs(n_samples=150, n_features=2, centers=3,
cluster_std=0.5, shuffle=True, random_state=0)

# Perform hierarchical clustering
agg_clustering = AgglomerativeClustering(n_clusters=3)
agg_clustering.fit(X)

# Plot the data points colored by their cluster
plt.scatter(X[:,0], X[:,1], c=agg_clustering.labels_,
cmap='rainbow')
plt.show()
```

This code uses the make_blobs function from the sklearn.datasets module to generate some sample data, then it performs hierarchical clustering on the data using the AgglomerativeClustering class from sklearn.cluster module, and finally, it plots the data points colored by their cluster.

You can adjust the number of clusters by changing the n_clusters parameter, and you can adjust the linkage method by passing linkage='ward', for example.

This is a simple example; you may need to preprocess and handle missing values and outliers in the real world.

# HIDDEN MARKOV MODEL (HMM)




| | |
|---|---|
| **Definition** | **Hidden Markov Model is a statistical Markov model in which the modeled system is assumed to be a Markov process with unobservable hidden states.** |
| **Main Domain** | **Structured Data, NLP & Speech Processing** |
| **Data Type** | **Reinforcement Learning** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Rewarding** |
| **Explainability** | **-** |

A Hidden Markov Model (HMM) is a statistical model used to describe a sequence of observations that are thought to have been generated by a series of underlying states. HMMs are widely used for speech recognition, bioinformatics, and natural language processing tasks.

An HMM consists of three main components:

- A set of states
- A set of observations
- A set of transition probabilities and observation probabilities

The states represent the underlying system or process that generates the observations, while the statements represent the data we can observe. The transition probabilities determine the probability of transitioning from one state to another, while the observation probabilities determine the probability of observing a particular data point given a specific form.

The key feature of an HMM is that the underlying state sequence is not directly observable, hence the name "hidden" Markov model. Instead, we can only observe the outputs, which are assumed to be conditionally independent given the underlying states.

The most common use of HMM is to infer the most likely sequence of hidden states given the observed data, this is known as the decoding problem, and it can be solved by using algorithms such as the Viterbi algorithm. Another everyday use of HMM is to estimate the model parameters given a set of observations; this is known as the learning problem.

In summary, A Hidden Markov Model (HMM) is a statistical model used to describe a sequence of observations that are thought to

have been generated by a sequence of underlying states. HMMs are widely used for tasks such as speech recognition, bioinformatics, and natural language processing.

The most common use of HMM is to infer the most likely sequence of hidden states given the observed data, this is known as the decoding problem, and it can be solved by using algorithms such as the Viterbi algorithm. Another everyday use of HMM is to estimate the model parameters given a set of observations; this is known as the learning problem.

Example:

Imagine we have a dataset of weather data, where each day is represented by an observation of either "sunny," "cloudy," or "rainy," and we want to use an HMM to predict the weather for the next day.

1. The first step is to initialize the HMM with a set of states, in this case, "sunny", "cloudy," and "rainy", and a bunch of observations, in this case, "sunny", "cloudy", and "rainy."
2. Next, we will estimate the transition probabilities between the states based on the historical weather data. For example, the probability of transitioning from "sunny" to "cloudy" might be 0.2, while the probability of transitioning from "cloudy" to "rainy" might be 0.5.
3. We will also estimate the observation probabilities, which determine the probability of observing a particular data point given a specific state. For example, the probability of observing "cloudy" given the state "sunny" might be 0.1, while the probability of observing "rainy" given the state "cloudy" might be 0.3.

4. Once we have estimated the model parameters, we can use the HMM to predict the weather for the next day.
5. For example, if the current weather is "sunny," the HMM might predict that the weather tomorrow will be "cloudy" with a probability of 0.2 or "sunny" with a probability of 0.8.

It's important to note that this is a simplified example, and in practice, using an HMM can be more complex and may require more data and computational resources. Additionally, HMMs can be extended to handle more complex cases, such as continuous observations, multiple observations, and multiple hidden states.

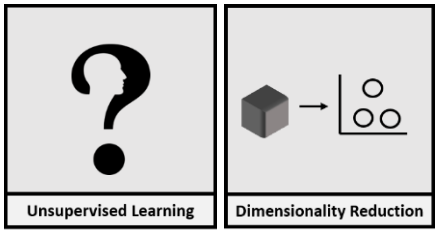
Code example of implementing a Hidden Markov Model (HMM) using Python's hmmlearn library. Here is an example of how to use the library to train a simple HMM to predict the weather based on two hidden states, **'sunny'** and **'rainy**,' and two observable states, **'walk'** and **'shop'**:

```
from hmmlearn import hmm
import numpy as np

# Define the model's parameters
states = ['sunny', 'rainy']
n_states = len(states)

observations = ['walk', 'shop']
n_observations = len(observations)

# Define the model's structure
model = hmm.MultinomialHMM(n_components=n_states)
model.startprob_ = np.array([0.6, 0.4])
model.transmat_ = np.array([[0.7, 0.3], [0.4, 0.6]])
model.emissionprob_ = np.array([[0.1, 0.4], [0.6, 0.3]])

# Define the sequence of observations
observation_sequence = np.array([[0, 1, 0]]).T

# Train the model
model = model.fit(observation_sequence)
```

```
# Predict the most likely sequence of hidden states
logprob, weather_sequence =
model.decode(observation_sequence, algorithm='viterbi')

# Print the results
print("Observation sequence: ", [observations[i] for i in
observation_sequence.ravel()])
print("Weather sequence: ", [states[i] for i in
weather_sequence])
```

This is a simple example of how to use hmmlearn to train a HMM, you could use this as a foundation to use HMM for more complex problems.

# INDEPENDENT COMPONENT ANALYSIS




| | |
|---|---|
| **Definition** | **ICA is a particular case of blind source separation. A typical example application is the "cocktail party problem" of listening in on one person's speech in a noisy room.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Dimension Reduction** |
| **Explainability** | **-** |

Independent Component Analysis (ICA) is a computational technique for separating a multivariate signal into independent non-Gaussian components. ICA identifies and extracts underlying independent sources from a mixture of observed signals. These sources are considered statistically independent from each other.

A typical application of ICA is to extract independent sources from a multichannel recording, such as an EEG or fMRI dataset. It is also used in image and video, natural language, and speech recognition, among other fields.

The basic idea of ICA is to identify a linear transformation of the original variables (e.g., signals) such that the new variables are as independent as possible. The transformed variables are called independent components (ICs). There are different algorithms to perform ICA, such as FastICA, JADE, and Infomax, among others. These algorithms estimate the underlying independent sources by maximizing the non-Gaussianity of the transformed variables or by minimizing mutual information or other measures of dependency between the transformed variables.

In summary, Independent Component Analysis (ICA) is a computational technique for separating a multivariate signal into independent non-Gaussian components. ICA identifies and extracts underlying independent sources from a mixture of observed signals. These sources are considered statistically independent from each other. A common application of ICA is to remove independent sources from a multichannel recording, such as an EEG or fMRI dataset. There are different algorithms to perform ICA, such as FastICA, JADE, and Infomax, among others. These algorithms estimate the underlying independent sources by maximizing the non-Gaussianity of the transformed variables or by minimizing mutual information or other measures of dependency between the transformed variables.

Example:

Imagine we have a recording of a concert that contains three different instruments: guitar, drums, and keyboard. The recording is a mixture of these three instruments, and we want to use ICA to separate the individual sources.

1. The first step is to preprocess the data by applying a linear mixing operation to the original signals of the instruments, resulting in a mixture of the signals.
2. Next, we will apply an ICA algorithm to the mixed signals, such as FastICA or Infomax, to estimate the independent components (ICs).
3. The algorithm will identify a linear transformation of the mixed signals such that the new variables are as independent as possible.
4. After the ICs are obtained, we can use a method called blind source separation (BSS) to separate the sources. This can be done by comparing the statistical properties of the ICs with the known properties of the original signals, such as the power spectrum.
5. Once the sources have been separated, we can listen to the instruments and hear how they sound separately.

It's important to note that this is a simplified example, and in practice, the process of using ICA can be more complex and may require more data and computational resources. Additionally, in real-world scenarios, the sources may be somewhat separate, and some residual mixture may be left. Also, the number of sources may not be known in advance, so the number of ICs obtained by the algorithm may need to be determined.

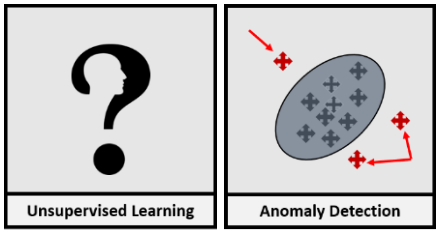
Example code of Independent Component Analysis (ICA) implemented in Python using the scikit-learn library:

```
from sklearn.decomposition import FastICA
import numpy as np

# Generate sample data
np.random.seed(0)
X = np.random.randn(200, 3)

# Fit the ICA model
ica = FastICA(n_components=3)
X_ica = ica.fit_transform(X)
```

In this example, we generate random data with 200 samples and three features, then fit an ICA model to it using the FastICA() class from scikit-learn. The n_components parameter is set to 3, indicating that we want to decompose the data into three independent components. The fit_transform() method is then used to find the separate components and return them as the X_ica variable.

Please keep in mind that this is a simple example for demonstration purposes, and in practice, you would use real-world data and experiment with different parameters and techniques to obtain the best results.

# ISOLATION FOREST




| | |
|---|---|
| **Definition** | **The isolation forest returns the anomaly score of each sample. It isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Anomaly Detection** |
| **Explainability** | **-** |

Isolation Forest (IF) is a machine-learning algorithm that can be used for anomaly detection. The algorithm is based on the concept of isolation, which is the idea that anomalies are data points that are different from the majority of the data points in the dataset.

IF builds a forest of decision trees, where each tree is trained on a random subset of the data. The decision tree is used to isolate observations by selecting a feature and then setting a split value between the maximum and minimum values of the selected feature. This process is repeated recursively until each tree leaf node represents a single observation.

The score of an observation is the number of splittings required to isolate it in the decision tree. The lower the score, the more abnormal the observation.

The following procedure identifies anomalies:

- The isolation forest algorithm is trained on the data set.
- Each data point is passed through the isolation forest.
- The number of splittings required to isolate the data point is calculated.
- The data points with a low number of splittings are considered anomalies.

IF is considered to be a robust algorithm for anomaly detection, mainly when the dataset is large, and it's able to identify anomalies with high accuracy and efficiency.

In summary, Isolation Forest (IF) is a machine-learning algorithm that can be used for anomaly detection. It's based on the concept of isolation which is the idea that anomalies are data points that are different from the majority of the data points in the dataset. IF builds a forest of decision trees, where each tree is trained on a

random subset of the data. The decision tree is used to isolate observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The score of an observation is the number of splittings required to isolate it in the decision tree. The lower the score, the more abnormal the observation. IF is considered a powerful algorithm for anomaly detection, particularly when the dataset is large, and it can identify anomalies with high accuracy and efficiency.

Example:

Imagine we have a dataset of network traffic data, where each data point represents a network connection, and we want to use IF to identify any abnormal connections.

1. The first step is preparing the data by removing irrelevant features and normalizing the values.
2. Next, we will create an instance of the IF model and fit it to the data.
3. We will then use the predict() method to predict the label of each data point as either "normal" or "anomalous."
4. The algorithm will assign a score to each data point, where a lower score indicates a higher likelihood of the connection being an anomaly.
5. We can then set a threshold on the score to determine which connections are considered anomalous.
6. We can then investigate the anomalous connections to determine whether they are malicious.

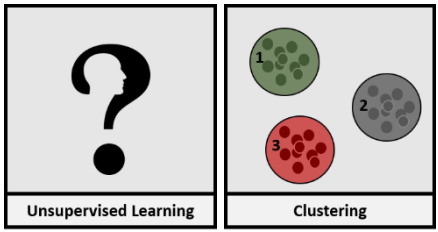
Code example of using the Isolation Forest algorithm in Python with the scikit-learn library:

```
from sklearn.ensemble import IsolationForest
import numpy as np

# Generate some random data
rng = np.random.RandomState(42)
X = rng.randn(100, 2)

# Create an instance of the IsolationForest class
clf = IsolationForest(random_state=rng)

# Fit the model to the data
clf.fit(X)

# Predict if each point is an outlier or not
y_pred = clf.predict(X)

# Print the results
print(y_pred)
```

In this example, the Isolation Forest algorithm is used to predict whether each point in the 2-dimensional dataset is an outlier. The **fit** method is used to train the model on the data, and the **predict** method is used to make predictions. **The y_pred** variable will contain the predicted labels, where 1 indicates that a point is not an outlier and -1 indicates that it is an outlier.

Please note that this is just an example, you need to adjust it to your specific needs and data, and it might only work as expected if you have the correct data.

# K-MEANS




| | |
|---|---|
| **Definition** | **K-means clustering is a method of vector quantification that aims to partition n observations into k clusters. Each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a cluster prototype.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

K-means is a widely used clustering algorithm in machine learning. It is a type of unsupervised learning algorithm that is used to find patterns or groupings in data without prior knowledge of the correct output. K-means aims to partition a set of points into K clusters, where each point belongs to the cluster with the nearest mean.

The algorithm works by initializing K centroids, randomly chosen points in the data, and then iteratively refining the clusters by reassigning each point to the cluster whose centroid it is closest to. The centroids are then recalculated as the mean of all the points assigned to that cluster. This process is repeated until the clusters no longer change or a maximum number of iterations is reached.

One of the advantages of the k-means algorithm is that it is computationally efficient and easy to implement, but it requires the number of clusters (K) to be specified in advance.

Example:

Imagine we have a dataset of customer data, where each data point represents a customer, and we want to use k-means to group similar customers together.

1. The first step is preparing the data by removing irrelevant features and normalizing the values.
2. Next, we will create an instance of the k-means model and fit it to the data.
3. We will then use the predict() method to assign a cluster label to each customer.
4. We can then examine the characteristics of each cluster to identify common patterns among the customers in that

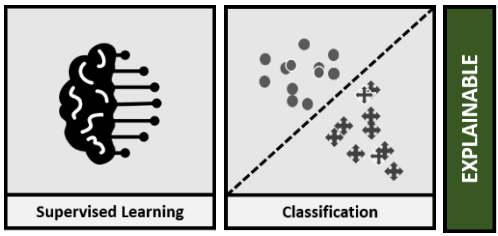
cluster. For example, customers in cluster 0 tend to be older and have higher incomes, while customers in cluster 1 are younger and have lower incomes.

Code example of K-means clustering implemented in Python using the scikit-learn library:

```
from sklearn.cluster import KMeans
import numpy as np

# Creating a sample dataset with 4 features and 150
samples
X = np.random.rand(150, 4)

# Initializing the KMeans algorithm with 3 clusters
kmeans = KMeans(n_clusters=3)

# Fitting the algorithm to the data
kmeans.fit(X)

# Accessing the cluster labels for each data point
labels = kmeans.labels_

# Accessing the coordinates of the cluster centers
cluster_centers = kmeans.cluster_centers_
```

In this example, the **KMeans** algorithm is imported from the **sklearn.cluster** library. A sample dataset is created using **numpy** with 150 samples and 4 features. The **KMeans** algorithm is initialized with n_clusters=3 and then fit to the data using the **fit()** method. The cluster labels for each data point can be accessed using the **labels_** attribute, and the coordinates of the cluster centers can be accessed using the **cluster_centers_** attribute.

# K-NEAREST NEIGHBOUR




| | |
|---|---|
| **Definition** | **K-Nearest Neighbor is a simple algorithm that stores all available cases and classifies the new data or cases based on a similarity measure. It is mostly used to classify a data point based on the classification of its neighbors.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Explainable** |

K-nearest neighbor (KNN) is a type of supervised machine learning algorithm that can be used for classification and regression tasks. The algorithm is based on the idea that similar data points tend to have similar labels and that the label of a new data point can be determined by looking at the labels of its nearest neighbors.

In the classification setting, the algorithm is trained on a labeled dataset. The training phase involves storing the training examples' feature values and class labels. Given a new input, the algorithm finds the k-nearest examples in the training set (based on some distance metric such as Euclidean distance) and assigns the input the most common class among its k-nearest neighbors.

In the regression setting, the algorithm works similarly to the classification setting, but instead of finding the most common class among its k-nearest neighbors, it finds the average value of the target variable among its k-nearest neighbors.

One of the advantages of the KNN algorithm is that it is simple to understand and implement, but it can be computationally expensive and sensitive to the choice of distance metric and the value of k.

Example:

1. Imagine we have a dataset of customer data, where each data point represents a customer, and the data includes the customer's age, income, and spending habits. We want to use KNN to predict which customers will most likely purchase a new product.
2. The first step is to prepare the data by removing any irrelevant features and normalizing the values.
3. Next, we will split the data into a training set and a test set.

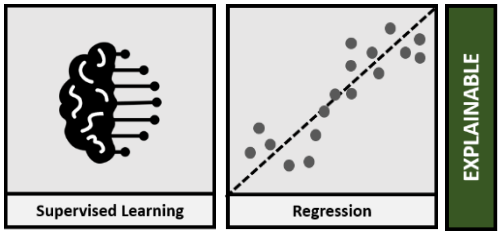
4. We will then create an instance of the KNN model and fit it to the training data using the fit() method.
5. We will then use the predict() method to predict the class labels of the test data.
6. We can then evaluate the model's performance using an appropriate evaluation metric, such as accuracy or F1-score.

Code example of using the k-nearest neighbors (KNN) algorithm for classification in Python using the scikit-learn library:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import make_classification

# Generate sample data
X, y = make_classification(n_samples=1000, n_features=4,
n_classes=2)

# Create and fit KNN model
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X, y)

# Predict on new data
X_new = [[-0.2, 0.4, 0.5, -0.1]]
print(knn.predict(X_new)) # [0]
```

This code uses the **make_classification** function to generate a sample dataset of 1000 samples and 4 features. The k-nearest neighbors classifier is then initialized with k=5 and fit to the data. The predict method is then used to predict the class of new unseen data.

# LINEAR REGRESSION




| | |
|---|---|
| **Definition** | **Linear regression attempts to model the relationship between two or more variables by fitting a linear equation to the observed data. One variable is considered the explanatory variable, and the other is regarded as the dependent variable. For example, a modeler might want to use a linear regression model to relate the weights of individuals to their height.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Regression** |
| **Explainability** | **Explainable** |

Linear regression is a statistical method used to model the relationship between a dependent variable (also known as the response variable) and one or more independent variables (also known as predictor variables or features). The goal of linear regression is to find the best-fitting straight line (or hyperplane in the case of multiple independent variables) that describes the relationship between the dependent and independent variables.

The basic assumption of linear regression is that the relationship between the independent variables and the dependent variable is linear, meaning that the change in the dependent variable is directly proportional to the change in the independent variables. The line of best fit is determined by minimizing the sum of the squared differences between the predicted and actual values.

Linear regression can be used for both simple linear regression (when there is one independent variable) and multiple linear regression (when there are numerous independent variables).

Example:

Imagine we have a dataset of housing prices, where each data point represents a house, and the data includes the house's size (in square feet), number of bedrooms, and age (in years). We want to use linear regression to predict the price of a house based on its size, number of bedrooms, and age.

1. The first step is preparing the data by removing irrelevant features and normalizing the values.
2. Next, we will split the data into training and test sets.
3. We will then create an instance of the linear regression model and fit it to the training data using the fit() method.

4. We will then use the predict() method to predict the price of the houses in the test data.
5. We can then evaluate the model's performance using an appropriate evaluation metric, such as mean squared error or R-squared

Code example of how to use linear regression with the scikit-learn library in Python:

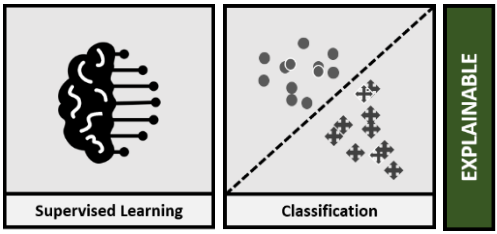
```
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

# Create a synthetic dataset
X, y = make_regression(n_samples=100, n_features=1,
noise=20)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Create a Linear Regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate the coefficient of determination (R^2)
r2 = model.score(X_test, y_test)

print("R^2:", r2)
```

This code first creates a synthetic dataset using the **make_regression** function from scikit-learn, then splits the data into training and test sets. Then, it creates a linear regression model from the **LinearRegression** class of scikit-learn, and fits the model to the training data using the **fit** method. Next, it makes

predictions on the test data using the **predict** method and calculates the coefficient of determination (R^2) using the **score** method, which measures the model's goodness of fit to the data. The output is the R^2, which measures how well the model fits the data. A value of 1.0 means that the model perfectly fits the data, while a value of 0.0 means that the model does not fit the data.

Please remember that this is just a simple example; in real-world scenarios, you should use cross-validation, feature scaling, and many other techniques to improve the model performance and avoid overfitting.

# LOGISTIC REGRESSION




| | |
|---|---|
| **Definition** | **Logistic regression is used to classify data by modeling the probability of having a particular class or event, such as pass/fail, won/lost, alive/dead, or healthy/sick. This can be extended to model multiple classes of events, e.g., to determine if an image contains a cat, dog, tiger, etc.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Explainable** |

Logistic regression is a statistical method for predicting a binary outcome (1/0, Yes/No, True/False) given a set of independent variables. It is a generalized linear model (GLM) type and uses a logistic function (also called the sigmoid function) to model a binary dependent variable.

In logistic regression, a model is developed to estimate the probability of the outcome being 1 (positive class) given a set of independent variables. The model is trained using a labeled dataset, where the outcome is known, and then it can be used to predict the outcome for new, unseen data. The predicted probability can then be thresholded to make a binary decision; for example, if the predicted probability is more significant than 0.5, the expected outcome is 1; otherwise, it is 0.

The logistic function is an S-shaped curve that maps any input value to a value between 0 and 1. This allows logistic regression to predict a probability of the outcome being 1 or 0.

Logistic regression is commonly used in various fields, such as medical research, social sciences, and marketing. It predicts a binary outcome based on one or multiple independent variables. Logistic regression can also be extended to multi-class classification problems by training multiple binary classifiers, one for each class.

It's important to note that logistic regression assumes that the relationship between the independent and dependent variables is logistic, and this assumption might not always hold in real-world scenarios. Therefore, checking the assumptions and interpreting the results with care is essential.

Example:

Imagine we want to predict whether a customer will buy a specific product based on their age, income and whether they are a homeowner. We have a dataset of customer information where the outcome (buy or not buy) is known.

1. The first step is to prepare the data by removing irrelevant features and normalizing the values if necessary.
2. Next, we will split the data into training and test sets.
3. We will then create an instance of the logistic regression model and fit it to the training data using the fit() method.
4. We will then use the predict_proba() method to predict the probability of customers buying the product in the test data.
5. We can then evaluate the model's performance using an appropriate evaluation metric, such as accuracy or AUC-ROC.

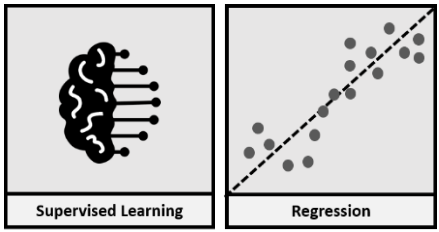
Code example of how to use logistic regression with the scikit-learn library in Python:

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Create a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10,
n_classes=2)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Create a logistic regression model
model = LogisticRegression()

# Fit the model to the training data
model.fit(X_train, y_train)
```

```
# Make predictions on the test data
y_pred = model.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

This code first creates a synthetic dataset using the **make_classification** function from scikit-learn, then splits the data into training and test sets. Then, it creates a logistic regression model from the **LogisticRegression** class of scikit-learn, and fits the model to the training data using the **fit** method. Next, it makes predictions on the test data using the **predict** method and calculates the accuracy of the model using the **accuracy_score** function from scikit-learn

# LSTM




| | |
|---|---|
| **Definition** | **Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in deep learning. Unlike standard feedforward neural networks, the LSTM has feedback connections. As a result, it can process individual data points (e.g., images) and entire data sequences (e.g., speech or video).** |
| **Main Domain** | **NLP & Speech Processing, Computer Vision** |
| **Data Type** | **Text, Image, Video** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Regression** |
| **Explainability** | **Not Explainable** |

Long Short-Term Memory (LSTM) is a Recurrent Neural Network (RNN) architecture designed to handle the problem of long-term dependencies in sequence data. RNNs are neural networks that process sequential data, such as time series, text, speech, or video. LSTMs are a specific type of RNN that can remember information for a longer time by using a memory cell, gates, and a hidden state.

A memory cell is a simple unit that stores information for an extended period. The gates are used to control the flow of information into and out of the memory cell. The hidden state is used to pass information from one time step to the next.

LSTMs are particularly useful for tasks such as language modeling, speech recognition, and machine translation, where the input data is a sequence of variable length, and the output depends on the entire input history.

Example:

Imagine we want to predict a company's stock prices based on historical data. We have a dataset of daily stock prices for the past five years.

1. The first step is to prepare the data by removing irrelevant features, normalizing the values, and splitting the data into sequences of a certain length (time steps) for input to the LSTM network.
2. Next, we will split the data into training and test sets.
3. We will then create an instance of the LSTM model using a library such as Keras or Tensorflow and add layers like LSTM, Dense, and Dropout layers to the model.

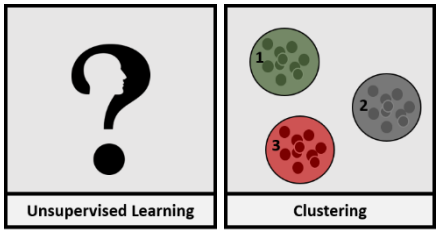
4. We will then train the model using the fit() method on the training data.
5. Finally, we will use the model to predict the stock prices on the test data.

Code example of how to use a Long Short-Term Memory (LSTM) network with the Keras library in Python:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM

# Generate a synthetic dataset
data = np.random.random((1000, 1, 1))
labels = np.random.randint(2, size=(1000, 1))

# Split the data into training and test sets
train_data = data[:800]
train_labels = labels[:800]
test_data = data[800:]
test_labels = labels[800:]

# Create a LSTM model
model = Sequential()
model.add(LSTM(32, input_shape=(1, 1)))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])

# Fit the model to the training data
model.fit(train_data, train_labels, epochs=10,
batch_size=32)

# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(test_data,
test_labels)

print("Test accuracy:", test_acc)
```

This code first generates a synthetic dataset using NumPy, then splits the data into training and test sets. Then, it creates a LSTM model using Keras's Sequential class, adding an LSTM layer and a dense output layer with a sigmoid activation function. Next, it compiles the model using the compile method, specifying the loss function as binary_crossentropy and optimizer as Adam. Then, it fits the model to the training data using the fit method for 10 epochs with batch_size of 32. Finally, it evaluates the model on the test data using the evaluate method and returns the loss and accuracy of the model on the test set.

# MEAN SHIFT




| | |
|---|---|
| **Definition** | **Mean shifting is a nonparametric feature space analysis procedure for finding the maxima of a density function, a so-called mode seeking algorithm.** |
| **Main Domain** | **Computer Vision** |
| **Data Type** | **Image, Video** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

Mean Shift is a clustering algorithm that aims to find a density function's mode (peak) in a given dataset. The algorithm assigns each data point to the cluster whose mean is closest to it. The mean is then recalculated for each cluster, and the process repeats until convergence.

Mean Shift is a non-parametric, density-based algorithm that does not require the number of clusters to be specified in advance. Instead, the algorithm works by iteratively shifting the centroids of the clusters until they converge to the modes of the density function. The algorithm stops when the centroids no longer move, or the change in centroid positions is below a certain threshold.

Mean Shift helps find clusters in datasets where the number and shape of clusters are not known in advance. It is also robust to noise and outliers and does not make assumptions about the data distribution like other clustering algorithms.

Example:

Imagine we have a dataset of customer locations, and we want to segment them into different regions based on their proximity. We can use mean Shift clustering to group customers who are located close to each other.

1. First, we will load and prepare the data by removing irrelevant features and normalizing the values if necessary.
2. Next, we will create an instance of the mean Shift model and fit it to the customer location data using the fit() method.
3. We will then use the predict() method to assign each customer to a cluster.

4. Finally, we can visualize the clusters on a map to understand the segmentation.

Code example of how to use mean shift clustering with the sklearn library in Python:

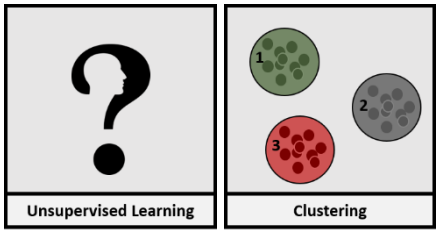
```
from sklearn.cluster import MeanShift
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generate synthetic data
X, _ = make_blobs(n_samples=1000, centers=[[1,1], [-1,-
1], [1,-1]], cluster_std=0.6)

# Create the mean shift model
ms = MeanShift(bandwidth=0.8)

# Fit the model to the data
ms.fit(X)

# Get the cluster labels
labels = ms.labels_

# Get the cluster centers
cluster_centers = ms.cluster_centers_

# Plot the data points, colored according to their
cluster
plt.scatter(X[:,0], X[:,1], c=labels)

# Plot the cluster centers
plt.scatter(cluster_centers[:,0], cluster_centers[:,1],
marker='x', c='black', s=200)

plt.show()
```

This code first generates a synthetic dataset of 3 clusters using the **make_blobs** function from sklearn.datasets, then it creates a mean shift model from the **MeanShift** class of sklearn.cluster. Next, it sets the bandwidth parameter to 0.8. Next, it fits the model to the data using the **fit** method. Then it gets the cluster

labels using the **labels_** attribute and cluster centers using the **cluster_centers_** attribute. Finally, it plots the data points colored according to their cluster and plots the cluster centers as black x.

Mean Shift Clustering is a density-based clustering algorithm that automatically finds the number of clusters by shifting points toward the mode of the density function of the data points. The bandwidth parameter controls the radius of the kernel used to smooth the density function. The larger the bandwidth, the more points will be considered in the density estimation, and the more clusters will be found.

# MOBILENET




| | |
|---|---|
| **Definition** | **MobileNets are based on a streamlined architecture that uses depth-wise separable convolutions instead of convolutions to build light wFeight deep neural networks.** |
| **Main Domain** | **Computer Vision, Classic Data Science, NLP & Speech Processing** |
| **Data Type** | **Image, Video, Text, Time Series, Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Clustering** |
| **Explainability** | **-** |

MobileNet is a lightweight convolutional neural network (CNN) designed to run efficiently on mobile and embedded devices, such as smartphones and tablets. MobileNet was developed by Google Research and is part of the TensorFlow project.

The architecture of MobileNet is based on depthwise separable convolutions, which reduces the number of parameters and computation required compared to traditional convolutional layers. This makes it more suitable for running on mobile devices with limited computational power and memory. MobileNet also uses a technique called "width multiplication," which allows for different model sizes to be created by adjusting the width of the layers.

MobileNet has been pre-trained on the ImageNet dataset and can be used for tasks such as image classification, object detection, and semantic segmentation. It can also be fine-tuned for specific tasks or retrained on new datasets.

Example:

Imagine we want to build a mobile app that can recognize different types of flowers. Then, we can use MobileNet to train a flower classification model on a dataset of flower images.

1. First, we will collect and prepare a dataset of images of different types of flowers and their labels.
2. Next, we will use MobileNet architecture, which is pre-trained on the ImageNet dataset, as the base for our model and fine-tune the model on our dataset of flower images.
3. We will then use the model to predict the class of a new image of a flower.

4. Finally, we can integrate the model into a mobile app and deploy it to users.

Code Example:

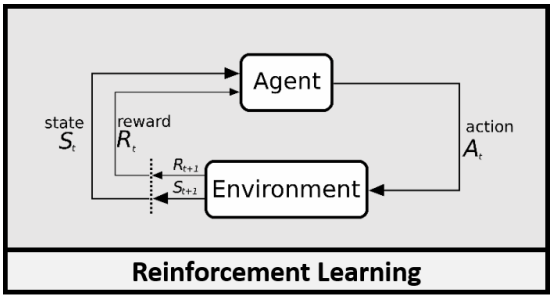
```
from keras.applications import MobileNet
from keras.layers import Dense, GlobalAveragePooling2D
from keras.models import Model
from keras.optimizers import Adam

# create the base pre-trained model
base_model = MobileNet(weights='imagenet',
include_top=False)

# add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)
# and a logistic layer
predictions = Dense(len(classes), activation='softmax')(x)

# this is the model we will train
model = Model(inputs=base_model.input,
outputs=predictions)

# first: train only the top layers (which were randomly
initialized)
# i.e. freeze all convolutional InceptionV3 layers
for layer in base_model.layers:
    layer.trainable = False

# compile the model (should be done *after* setting layers
to non-trainable)
model.compile(optimizer=Adam(),
loss='categorical_crossentropy')

# train the model on the new data for a few epochs
model.fit(X_train, y_train, epochs=10, batch_size=32)
```

This code first imports the necessary libraries, creates an instance of the MobileNet model with pre-trained weights on the ImageNet dataset, adds a global spatial average pooling layer and a logistic layer, and fine-tunes the model on our dataset of flower

images by freezing all the convolutional layers and training only the last layers. Finally, the model is trained on the new data for several epochs.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. MobileNet is a deep learning model that requires a large amount of data to train and can be computationally expensive. Therefore, the final performance of the model will also depend on the quality of the data and the tuning of the hyperparameters.

# MONTE CARLO ALGORITHM




| | |
|---|---|
| **Definition** | **A Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a particular (typically small) probability.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Reinforcement learning** |
| **Learning Paradigm** | **Rewarding** |
| **Explainability** | **-** |

The Monte Carlo algorithm is a statistical method for approximating a solution to a problem by generating random samples from a probability distribution. The algorithm is named after the Monte Carlo casino in Monaco, where randomness is a key feature of gambling games.

The Monte Carlo algorithm can be used to solve a wide range of problems, such as simulating physical systems, solving optimization problems, and estimating the value of mathematical expressions. The basic idea behind the algorithm is to use random sampling to estimate the value of an unknown quantity.

The Monte Carlo algorithm has several key steps:

- Define the problem and the solution space.
- Generate random samples from a probability distribution that represents the solution space.
- Use the samples to estimate the value of the unknown quantity.
- Repeat the process multiple times to reduce the uncertainty in the estimate.

Here is an example of how the Monte Carlo algorithm might be used to estimate the value of pi:

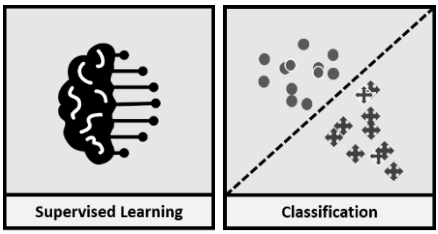
```
import random

# Number of darts to throw
n = 10000

# Number of darts that fall inside the circle
hits = 0

# Throw darts
for i in range(n):
    x = random.random()
    y = random.random()
    if x*x + y*y <= 1.0:
```

```
        hits += 1

# Calculate the value of pi
pi = 4.0 * hits / n
```

This code throws darts at a unit square that circumscribes a unit circle. The ratio of the number of darts that fall inside the circle to the total number of darts thrown approximates the ratio of the area of the circle to the area of the square. Multiplying this ratio by 4 gives an estimate of pi.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Additionally, the Monte Carlo algorithm requires a large amount of random samples to be generated to reduce the uncertainty in the estimate. The final performance of the algorithm will also depend on the data quality and the hyperparameters tuning.

# MULTIMODAL PARALLEL NETWORK




| | |
|---|---|
| **Definition** | **A Multimodal Parallel Network helps to manage audio-visual event localization by processing both audio and visual signals simultaneously.** |
| **Main Domain** | **Computer Vision, Speech Processing** |
| **Data Type** | **Video** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **-** |

Multimodal parallel networks (MPNs) are a class of deep learning models that are designed to handle multiple types of input data, such as images, text, and audio. They allow the integration of information from numerous sources and use the complementary information provided by each modality to improve the model's performance.

The basic architecture of an MPN consists of multiple parallel branches, each of which is responsible for processing a specific data modality. Each branch typically has its own set of layers that are optimized for the particular type of data it is processing. The outputs from each branch are then combined and processed by a shared set of layers responsible for making the final prediction.

MPNs have been used for various tasks, such as image and text classification, image captioning, and video understanding.

Here is an example of how an MPN might be used for a visual question-answering task:

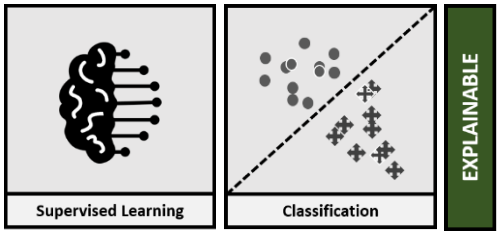
```
from keras.layers import Input, Dense, LSTM
from keras.models import Model

# Define inputs for image and question
image_input = Input(shape=(224, 224, 3))
question_input = Input(shape=(None,))

# Define image branch
x = ... (image processing layers, such as convolutional
layers)

# Define question branch
y = ... (question processing layers, such as an LSTM layer)

# Combine the output of the two branches
z = ... (combining layers, such as a concatenation layer)

# Define the final prediction layers
output = ... (fully connected layers)
```

```
# Create the model
model = Model(inputs=[image_input, question_input],
outputs=output)
```

This code first defines two separate input layers for the image and the question and then defines two different branches for processing each modality. One branch is for image processing, typically containing convolutional layers, and the other is for question processing and typically contains LSTM layers. The output of both branches is then combined and processed by a shared set of layers responsible for making the final prediction.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Additionally, MPNs are deep learning models and require a large amount of data to train, and can be computationally expensive. Therefore, the final performance of the model will also depend on the data quality and the hyperparameters' tuning.

# NAIVE BAYES CLASSIFIERS




| | |
|---|---|
| **Definition** | **Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying the Bayes theorem with strong (naive) independence assumptions between features.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Explainable** |

Naive Bayes classifiers are a family of probabilistic algorithms based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Furthermore, they are probabilistic, which means they calculate the probability of each class for a given input and then output the class with the highest probability.

Naive Bayes classifiers are widely used in text classification, spam filtering, sentiment analysis, and medical diagnosis. They are simple, fast, and easy to implement, which makes them a popular choice for many applications.

There are different types of Naive Bayes classifiers, each with an other probability distribution assumption. The three most commonly used are:

- Gaussian Naive Bayes assumes that the continuous features follow a normal distribution.
- Multinomial Naive Bayes: used for discrete data such as text and counts the number of occurrences of each feature per class.
- Bernoulli Naive Bayes: similar to Multinomial Naive Bayes but only considers whether a feature is present.

Code example of how a Gaussian Naive Bayes classifier might be used for a simple binary classification problem:

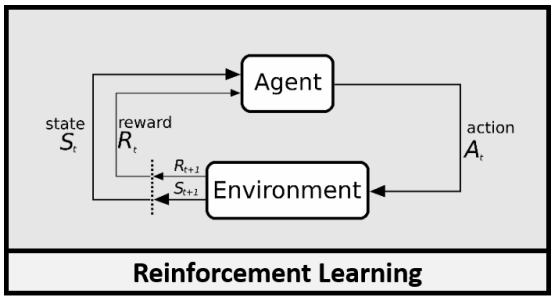
```
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import make_classification

# Generate a sample dataset
```

```
X, y = make_classification(n_samples=100, n_features=2,
n_informative=2, n_redundant=0)

# Create a Gaussian Naive Bayes classifier
clf = GaussianNB()

# Fit the classifier to the data
clf.fit(X, y)

# Predict the class of new data
pred = clf.predict([[0, 0], [1, 1]])

# Output the predicted class
print(pred)
```

This code first generates a sample dataset, creates a Gaussian Naive Bayes classifier, fits the classifier to the data, predicts the new data class, and outputs the predicted class.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Also, Naive Bayes classifiers are based on probabilistic assumptions and can be sensitive to irrelevant features and data. The final performance of the model will also depend on the data quality and the hyperparameters tuning.

# PROXIMAL POLICY OPTIMIZATION




**Reinforcement Learning**

| | |
|---|---|
| **Definition** | **A family of policy gradient methods for Reinforcement Learning that alternate between sampling data and optimizing a surrogate objective function using stochastic gradient ascent.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data, Time Series** |
| **Data Environment** | **Reinforcement Learning** |
| **Learning Paradigm** | **Rewarding** |
| **Explainability** | **-** |

Proximal Policy Optimization (PPO) is an algorithm for training reinforcement learning (RL) agents. It is a variant of the popular trust region policy optimization (TRPO) algorithm. PPO aims to improve the stability and efficiency of the RL training process by using a "proximal" objective function that is designed to be similar to the original objective function but is easier to optimize.

The key idea behind PPO is to use a "clip" function to limit the change in the policy probability distribution, which helps to ensure that the updated policy remains similar to the original policy. This helps to stabilize the training process and prevent the agent from moving too far away from the optimal solution.

PPO also uses an adaptive learning rate schedule and an adaptive value function regularization, which helps improve the training process's efficiency.

Here is an example of how PPO might be used to train an agent to play a simple game:

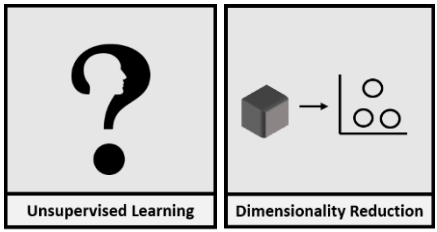
```
import gym
import torch.optim as optim
from ppo import PPO

# Create the environment
env = gym.make('CartPole-v0')

# Create the agent
agent = PPO(env.observation_space.shape[0],
env.action_space.n)

# Create the optimizer
optimizer = optim.Adam(agent.parameters(), lr=1e-3)

# Train the agent
for i_episode in range(1000):
    obs = env.reset()
    done = False
    while not done:
        action, _ = agent.act(obs)
```

```
            obs, reward, done, _ = env.step(action)
            agent.put_data((obs, action, reward, done))
        agent.update(optimizer)
```

This code first creates an environment using OpenAI Gym's CartPole-v0 environment, makes an agent using the PPO algorithm, creates an optimizer, and then trains the agent by repeatedly running episodes of the game, collecting data from the environment, and updating the agent's parameters using the optimizer.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Additionally, PPO is a complex algorithm and requires a large amount of data to train, and can be computationally expensive. Therefore, the final performance of the model will also depend on the quality of data and the tuning of the hyperparameters.

# PRINCIPAL COMPONENT ANALYSIS




| | |
|---|---|
| **Definition** | **The basic idea of principal component analysis (PCA) is to reduce the dimensionality of a data set consisting of many variables that are either strongly or slightly correlated with each other while preserving as much as possible the variation present in the data set.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **Dimension Reduction** |
| **Explainability** | **-** |

Principal Component Analysis (PCA) is a dimensionality reduction technique used to project high-dimensional data onto a lower-dimensional space while retaining as much of the original variance as possible. It is a linear technique that is commonly used for data visualization, noise reduction, and feature extraction.

The basic idea behind PCA is to find the directions in the feature space with the highest variance in the data and represent them as a new set of uncorrelated variables called principal components. These main components can be used as a new set of features for further analysis or modeling.

The algorithm of PCA can be summarized as follows:

1. Standardize the data by subtracting the mean and scaling by the standard deviation
2. Compute the covariance matrix of the data
3. Compute the eigenvectors and eigenvalues of the covariance matrix
4. Select the k eigenvectors that correspond to the k largest eigenvalues to form a k x d-dimensional matrix
5. Use this matrix to transform the data into a k-dimensional space

Here is an example of how PCA might be used to reduce the dimensionality of a dataset:

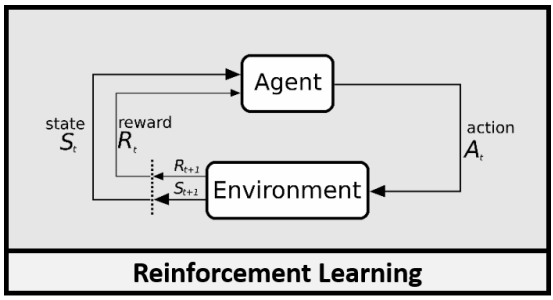
```
from sklearn.decomposition import PCA
import numpy as np

# Create a sample dataset
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1],
[3, 2]])

# Create a PCA model with 2 components
pca = PCA(n_components=2)

# Fit the model to the data
```

```
pca.fit(X)

# Transform the data
X_pca = pca.transform(X)

# Output the transformed data
print(X_pca)
```

This code first creates a sample dataset, creates a PCA model with two components, fits the model to the data, and then transforms the data into a 2-dimensional space.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Additionally, PCA is a linear technique and may not work well for non-linear data or for capturing complex relationships between features.

# Q-LEARNING




**Reinforcement Learning**

| | |
|---|---|
| **Definition** | **Q-learning is a model-free reinforcement learning algorithm for learning the value of an action in a given state. It does not require a model of the environment.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data, Time Series** |
| **Data Environment** | **Reinforcement Learning** |
| **Learning Paradigm** | **Rewarding** |
| **Explainability** | **-** |

Q-learning is a model-free, off-policy reinforcement learning (RL) algorithm. It is used to learn the optimal action-value function, also known as the Q-function, that describes the expected return for taking a specific action in a particular state and following a certain policy.

The Q-function is defined as Q(s, a) = E[R(t) | S(t) = s, A(t) = a], where s is the state, a is the action, R(t) is the reward, and E[.] denotes the expected value. The goal of Q-learning is to find the optimal Q-function, Q*(s, a) = max_π E[R(t) | S(t) = s, A(t) = a, π], where π is the policy.

The Q-learning algorithm can be summarized as follows:

1. Initialize the Q-function with arbitrary values
2. For each episode: a. Initialize the current state b. For each step of the episode: i. Select an action using an exploration strategy, such as epsilon-greedy ii. Take action and observe the next state, reward, and whether the episode is terminated iii. Update the Q-function using the observed information and the Bellman equation: Q(s, a) = Q(s, a) + α (r + γ * max_a(Q(s', a)) - Q(s, a)) c. Repeat step b until the episode is terminated
3. Repeat step 2 until the Q-function converges.

Here is an example of how Q-learning might be used to train an agent to play a simple game:

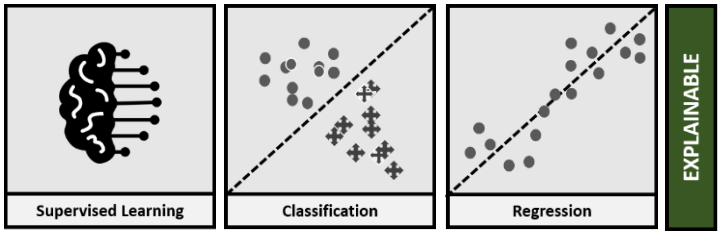
```
import gym
import numpy as np

# Create the environment
env = gym.make('FrozenLake-v0')

# Initialize the Q-function
Q = np.zeros((env.observation_space.n,
env.action_space.n))
```

```
# Define the learning parameters
alpha = 0.8
gamma = 0.95
epsilon = 0.1

# Train the agent
for i_episode in range(1000):
    obs = env.reset()
    done = False
    while not done:
        # Select an action using epsilon-greedy
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q[obs, :])
        # Take the action and observe the next state,
reward, and whether the episode is terminated
        next_obs, reward, done, _ = env.step(action)
        # Update the Q-function
        Q[obs, action] = Q[obs, action] + alpha * (reward
+ gamma * np.max(Q[next_obs, :]) - Q[obs, action])
        obs = next_obs
```

This code first creates an environment using OpenAI Gym's FrozenLake-v0 environment, initializes the Q-function, defines the learning parameters, and then trains the agent by repeatedly running episodes of the game, selecting actions using epsilon-greedy, taking action and observing the next state, reward, and whether the episode is terminated, and updating the Q-function.

It's important to note that this is a simple example, and you may need to adjust.

# RANDOM FORESTS




| | |
|---|---|
| **Definition** | **Random forests are an ensemble learning method that operates by constructing many decision trees at training time and outputting the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured Data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification, Regression** |
| **Explainability** | **Explainable** |

Random Forests is an ensemble learning method for classification and regression problems. It is a collection of decision trees, where each tree is trained on a random subset of the data and a random subset of the features. The final prediction is made by taking the average or mode of the predictions of all the trees in the forest.

The basic idea behind random forests is to reduce the variance and increase the bias of the individual decision trees by averaging the predictions of many uncorrelated trees. This helps to improve the generalization performance of the model and reduce overfitting.

The algorithm of Random Forest can be summarized as follows:

1. For each tree in the forest: a. Select a random subset of the data with replacement, also known as bootstrapping. b. Select a random subset of features. c. Train a decision tree on this subsample and features.
2. For classification problems, take the majority vote of the predictions of all the trees; for regression, take the average of the predictions.

Here is an example of how Random Forest might be used to train a classifier on a dataset:

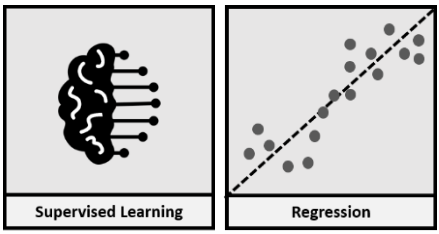
```
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Load a sample dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Create a random forest classifier
clf = RandomForestClassifier(n_estimators=100,
random_state=0)

# Fit the model to the data
clf.fit(X, y)
```

```
# Predict on a new sample
x_new = [[5, 3.2, 1.2, 0.2]]
print(clf.predict(x_new))
```

This code first loads a sample dataset creates a random forest classifier, fits the model to the data, and then makes a prediction on a new sample.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Random Forest is a robust algorithm that can handle high-dimensional data and missing values, but it can be computationally expensive. The final performance of the model will also depend on the data quality and the hyperparameters tuning.

# RECURRENT NEURAL NETWORK




| | |
|---|---|
| **Definition** | **A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit material dynamic behavior.** |
| | **Computer Vision, NLP & Speech processing** |
| **Main Domain** | **Computer Vision, NLP & Speech Processing** |
| **Data Type** | **Time series, Text, Image, Video** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Regression** |
| **Explainability** | **Not Explainable** |

A recurrent neural network (RNN) is a type of neural network that can process sequential data, such as time series or natural language. RNNs have a "memory" of hidden states that can store information from previous time steps, allowing them to understand the context and make predictions based on previous inputs. This makes them well-suited for language modeling, speech recognition, and video analysis tasks. In addition, RNNs can be unrolled through time to form a feedforward neural network with shared weights, also called a deep RNN or deep recurrent network.

One example of a task where an RNN would be helpful is language translation. In this task, the RNN would take in a sentence in one language (such as English) as input and output a translation of that sentence in another language (such as French). The RNN would use its hidden states to keep track of the meaning of the sentence as it processes each word, allowing it to generate a more accurate translation.

Another example is speech recognition, where an RNN can be used to transcribe spoken words into text. The RNN would process the audio data over time and use its hidden states to understand the context of the spoken words, allowing it to transcribe the speech more accurately.

Another example is stock price prediction, where an RNN can be used to predict the future prices of a stock based on past prices and other market data. The RNN would use its hidden states to understand the trends in the market data and make predictions accordingly.

These are just a few examples of the types of tasks that RNNs can be used for, but they can also be applied to many other problems.

An example of how you might implement an RNN in Python using the Keras library:

```
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense

# define the model
model = Sequential()
model.add(SimpleRNN(32, input_shape=(None, 1)))
model.add(Dense(1))

# compile the model
model.compile(optimizer='rmsprop', loss='mse')

# generate some dummy data
import numpy as np
x_train = np.random.random((1000, 10, 1))
y_train = np.random.random((1000, 1))

# train the model
model.fit(x_train, y_train, epochs=10)
```

In this example, we're using the SimpleRNN layer from Keras to define our RNN. The SimpleRNN layer takes an input of shape (batch_size, timesteps, input_dim) and returns an output of shape (batch_size, output_dim). In this example, we're using single input and output dimensions, but in practice, you could use more. The input_shape argument specifies the shape of the input data, and in this case, we're using None for the timesteps dimension so that the model can accept input sequences of any length.

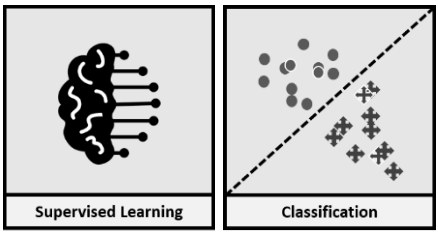
After defining the model, we use the compile method to specify the optimizer and loss function we want. We use the rmsprop optimizer and the mean squared error (MSE) loss in this case.

Finally, we're generating some dummy data using numpy and the fit method to train the model on this data. The fit method takes in

the training data (x_train and y_train) and the number of epochs to train for.

This is a basic example; in practice you should preprocess the data and tweak other hyperparameters for the best result.

# RESNET




| | |
|---|---|
| **Definition** | **A residual neural network (ResNet) is an artificial neural network (ANN) that builds on constructs known from pyramidal cells in the cerebral cortex by using jump connections or shortcuts to skip some layers.** |
| **Main Domain** | **Computer Vision** |
| **Data Type** | **Image** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Not Explainable** |

ResNet (Residual Network) is a type of deep neural network architecture that allows for training very deep networks by addressing the issue of vanishing gradients. In addition, ResNet uses skip connections, or shortcuts, to allow the gradient to bypass one or more layers, enabling the network to learn more complex features and improve performance. This architecture was introduced in the paper "Deep Residual Learning for Image Recognition" by He et al. (2016). ResNet has been used in various computer vision tasks and has achieved state-of-the-art results on image classification benchmarks such as ImageNet.

An example of how you might implement a ResNet architecture in Python using the Keras deep learning library:

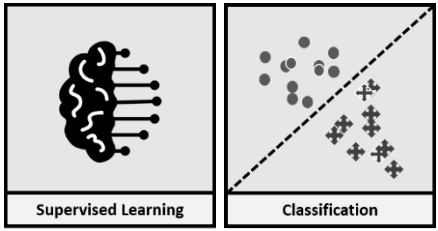
```
from keras.layers import Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization, Flatten, Conv2D, AveragePooling2D
from keras.models import Model

def resnet_block(inputs, num_filters, kernel_size, strides, activation='relu'):
    x = Conv2D(num_filters, kernel_size, strides=strides, padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    x = Conv2D(num_filters, kernel_size, strides=strides, padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)
    return x

def ResNet(input_shape=(32,32,3), classes=10):
    inputs = Input(shape=input_shape)
    x = ZeroPadding2D((3,3))(inputs)
    x = Conv2D(16, (7,7), strides=(1,1))(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = ZeroPadding2D((1,1))(x)
    x = MaxPooling2D((3,3), strides=(2,2))(x)
```

```
    # stack residual blocks
    for i in range(3):
        strides = (1,1) if i==0 else (2,2)
        x = resnet_block(x, 64, 3, strides)

    x = AveragePooling2D((8,8))(x)
    x = Flatten()(x)
    x = Dense(classes, activation='softmax')(x)

    model = Model(inputs, x, name='ResNet')
    return model

model = ResNet()
model.summary()
```

This example is a simplified version of ResNet, it's a good start for understanding the structure of ResNet, but to get better accuracy, you can use the original paper's architecture.

Note that this is just an example, and the architecture and parameters used in this example may not be optimal for a specific task or dataset. It's also important to note that this code snippet is just an example, and the final model might need further fine-tuning and training.

# SPATIAL TEMPORAL GRAPH CONVOLUTIONAL NETWORKS




| | |
|---|---|
| **Definition** | **Spatial-Temporal Graph Convolutional Networks is a convolutional neural network that automatically learns spatial and temporal patterns from data.** |
| **Main Domain** | **Computer Vision** |
| **Data Type** | **Video** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **-** |

Spatial-Temporal Graph Convolutional Networks (STGCN) is a type of deep learning model that can be used to analyze graph-structured data. STGCNs use convolutional neural networks (CNNs) to extract features from the spatial-temporal data and graph convolutional layers to capture the relationships between the data points. This allows the model to learn the spatial-temporal patterns in the data and make predictions or classify the data. STGCNs have been used in various applications, such as traffic forecasting, human action recognition, and weather prediction.

An example of how to implement a simple Spatial-Temporal Graph Convolutional Network (STGCN) in Python using the deep learning library Keras:

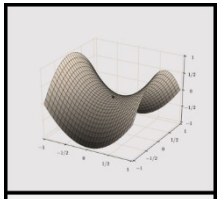
```
from keras.layers import Input, Convolutional, Reshape, Dense, Flatten
from keras.models import Model

# Define input with shape (batch_size, time_steps, num_nodes, num_features)
input_data = Input(shape=(time_steps, num_nodes, num_features))

# Apply 1D convolutional layers to extract spatial-temporal features
conv1 = Convolutional(64, kernel_size=1, activation='relu')(input_data)
conv2 = Convolutional(64, kernel_size=1, activation='relu')(conv1)

# Flatten the output of the convolutional layers
flatten = Flatten()(conv2)

# Apply fully connected layers to classify the data
fc1 = Dense(64, activation='relu')(flatten)
output = Dense(num_classes, activation='softmax')(fc1)

# Create the STGCN model
```

```
model = Model(inputs=input_data, outputs=output)

# Compile the model with a loss function and optimizer
model.compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])

# Train the model on the input data and labels
model.fit(input_data, labels, batch_size=32, epochs=10)
```

Please keep in mind that this is a straightforward example, and in practice, you will need to fine-tune the architecture of the network, the number of layers and neurons, the activation functions, and the hyperparameters of the optimizer based on the characteristics of your specific dataset.

Also, note that this code snippet doesn't include the graph convolutional layers, which are the core of STGCN; they are usually implemented using the chebyshev polynomials or GCN layers; you can use those layers to enhance the performance of your network.

# STOCHASTIC GRADIENT DESCENT



Optimization

| | |
|---|---|
| **Definition** | **Stochastic gradient descent is an iterative method for optimizing an objective function with suitable smoothing properties. It can be considered as a stochastic approach to gradient descent optimization. It involves replacing the actual gradient (computed from the entire data set) with an estimate (calculated from a randomly selected subset of the data).** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured data** |
| **Data Environment** | **-** |
| **Learning Paradigm** | **Optimization** |
| **Explainability** | **-** |

Stochastic Gradient Descent (SGD) is an optimization algorithm used to minimize a given function, typically the loss function of a machine learning model. The algorithm updates the parameters of the model in the direction of the negative gradient of the function concerning the parameters, intending to reduce the value of the function.

The critical feature of SGD is that it processes training examples one at a time or in small groups called mini-batches rather than the entire dataset. This makes the algorithm more computationally efficient and allows it to progress even when the dataset is too large to fit into memory.

SGD is widely used in machine learning and deep learning, in particular, to optimize the parameters of neural networks. It is a first-order optimization algorithm, meaning it only considers the gradient of the function with respect to the parameters and not the second-order information such as the Hessian matrix.

SGD has several variants, such as:

- Mini-batch Gradient Descent: This is a variant of stochastic gradient descent where instead of using one sample at a time, it uses a small batch of samples in each iteration.
- Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.
- Adagrad: it adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.
- Adadelta: it also adapts the learning rate per parameter and does not require manual tuning of a learning rate.
- Adam: it combines the advantages of both Adagrad and RMSProp.

It is important to note that the choice of optimizer depends on the problem and the dataset, and there is no one-size-fits-all solution.

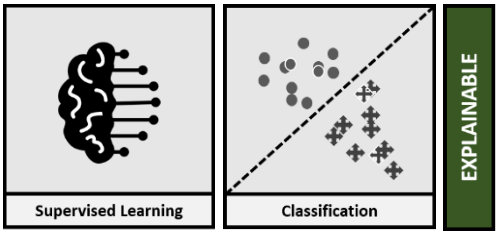
An example of how to use the Stochastic Gradient Descent (SGD) optimizer in Python using the deep learning library Keras:

```
from keras.optimizers import SGD
from keras.models import Sequential
from keras.layers import Dense

# Create a simple model
model = Sequential()
model.add(Dense(64,                input_dim=num_features,
activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# Compile the model with the SGD optimizer
sgd     =     SGD(lr=0.01,     decay=1e-6,     momentum=0.9,
nesterov=True)
model.compile(loss='categorical_crossentropy',
optimizer=sgd, metrics=['accuracy'])

# Train the model on the input data and labels
model.fit(input_data, labels, batch_size=32, epochs=10)
```

In this example, the learning rate ('**lr**') is set to 0.01, which means that the model parameters will be updated by subtracting 0.01 times the gradient of the loss function concerning the parameters. Furthermore, the '**decay**' parameter is used to reduce the learning rate over time, '**momentum**' parameter is used to accelerate SGD in the relevant direction, and dampens oscillations '**nesterov**' is used to give a slight variant to the standard momentum.

It's worth noting that you can also use other optimizers, such as Adam or Adadelta, which are also provided by Keras and can be used similarly. It's important to note that the choice of optimizer depends on the problem and the dataset, and there is no one-size-fits-all solution.

# SUPPORT VECTOR MACHINE




| | |
|---|---|
| **Definition** | **SVM are linear classifiers based on the principle of margin maximization. They perform the classification task by constructing the hyperplane in a higher-dimensional space that optimally separates the data into two categories.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification** |
| **Explainability** | **Explainable** |

Support Vector Machine (SVM) is a supervised learning algorithm that can be used for both classification and regression problems. SVM aims to find the best boundary (or hyperplane) that separates the different classes in a high-dimensional space. The limit is chosen to maximize the margin, which is the distance between the boundary and the closest data points of each class, also known as the support vectors.

The basic idea behind SVM is to transform the original data into a higher-dimensional space (also known as feature space) using a kernel function and then find the linear boundary that separates the classes in this space. The kernel function can be chosen based on the problem and the type of data, and it is used to map the data into a space where it is linearly separable.

The algorithm of SVM can be summarized as follows:

1. Transform the data into a higher-dimensional space using a kernel function.
2. Find the linear boundary that separates the classes by maximizing the margin.
3. Use this boundary to classify new data points.
4. Here is an example of how SVM might be used to train a classifier on a dataset:

```
from sklearn import datasets
from sklearn.svm import SVC

# Load a sample dataset
iris = datasets.load_iris()
X, y = iris.data[:, :2], iris.target

# Create a support vector classifier
clf = SVC(kernel='linear', C=1)

# Fit the model to the data
clf.fit(X, y)
```

```
# Predict on a new sample
x_new = [[5, 3.2]]
print(clf.predict(x_new))
```

This code first loads a sample dataset creates a support vector classifier, fits the model to the data, and then makes a prediction on a new sample.

It's important to note that this is a simple example, and you may need to adjust the parameters or use different libraries or techniques depending on the specific problem you are trying to solve. Additionally, SVM can handle high-dimensional data and non-linear boundaries, but it can be sensitive to kernel function choice and the hyperparameters' tuning.

# WAVENET


| | |
|---|---|
| **Definition** | **Wavenet is a deep neural network for generating raw audio waveforms. The model is fully probabilistic and autoregressive, where the predictive distribution for each audio sample depends on all previous ones.** |
| **Main Domain** | **NLP & Speech Processing** |
| **Data Type** | **Time Series** |
| **Data Environment** | **Unsupervised Learning** |
| **Learning Paradigm** | **NLP Task** |
| **Explainability** | **-** |

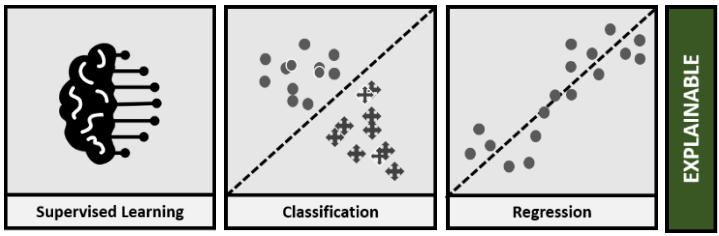
WaveNet is a deep neural network architecture for generating raw audio waveforms. It was developed by Google's DeepMind team and introduced in 2016. WaveNet uses a variant of the convolutional neural network (CNN) architecture and is trained on a dataset of audio waveforms. It can generate high-quality, realistic-sounding speech and music by modeling the underlying probability distributions of the audio signal. WaveNet is also used for Text-to-Speech (TTS) and music synthesis.

Example:

To use WaveNet for text-to-speech, you would first need to train the model on a dataset of audio waveforms and corresponding speech transcriptions. This can be done using a technique called "teacher forcing," where the model is provided with the correct output (the audio waveform) at each time step during training. Once the model is trained, you can then use it to generate audio waveforms from new input text. This is typically done by feeding the input text through an encoder to convert it into a compact representation that can be used as the initial state of the WaveNet model.

For music synthesis, you must train the model on a dataset of audio waveforms and corresponding MIDI notes or other musical information. Once the model is trained, you can generate audio waveforms from new input musical information. It's worth noting that training a WaveNet model requires a significant amount of computational resources, and it may take several days to several weeks to train a WaveNet model, depending on the model's complexity and the training dataset's size.

# XGBOOST




| | |
|---|---|
| **Definition** | **XGBoost is an extension of gradient boosted decision trees (GBM) and is specifically designed to improve speed and performance by using regularization methods to combat overfitting.** |
| **Main Domain** | **Classic Data Science** |
| **Data Type** | **Structured data** |
| **Data Environment** | **Supervised Learning** |
| **Learning Paradigm** | **Classification, Regression** |
| **Explainability** | **Explainable** |

XGBoost is an open-source software library for gradient boosting on decision trees. It stands for "Extreme Gradient Boosting" and is a powerful tool for building machine learning models, particularly in structured data such as classification and regression problems.

The library provides an efficient implementation of the gradient boosting algorithm, which is a method that combines multiple weak models to form a strong model. It uses decision trees as base models and iteratively adds new trees to correct the mistakes made by previous trees. XGBoost is known for its performance and computational efficiency and is widely used in machine-learning competitions and real-world applications.

Its features are particularly useful, such as support for parallel and distributed computing, efficient handling of missing values, and built-in handling of categorical variables. It also provides a rich set of parameters that can be used to fine-tune the model's performance.

In summary, XGBoost is an optimized and distributed gradient boosting library designed to be highly efficient and scalable and works well with structured data.

An example of how to use XGBoost for a binary classification problem in Python:

```
import xgboost as xgb
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Create a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10,
n_classes=2)

# Split the data into training and test sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Create an XGBoost data matrix from the training data
dtrain = xgb.DMatrix(X_train, label=y_train)

# Create an XGBoost data matrix from the test data
dtest = xgb.DMatrix(X_test, label=y_test)

# Define the parameter dictionary for the model
params = {'objective': 'binary:logistic', 'max_depth': 2}

# Train the model using the training data
model = xgb.train(params, dtrain)

# Make predictions on the test data
y_pred = model.predict(dtest)

# Convert the predicted probabilities to binary class
labels
y_pred_class = (y_pred > 0.5).astype(int)

# Compare the predicted class labels to the true class
labels
accuracy = (y_pred_class == y_test).mean()

print("Accuracy:", accuracy)
```

This code first creates a synthetic dataset using the make_classification function from scikit-learn, then splits the data into training and test sets. Then, it converts the training and test sets into XGBoost's data matrix format, which is more efficient for training and prediction. Next, it defines the parameters for the model; the objective is set to binary: logistic, and max_depth is set to 2. Then, it trains the model using the training data and makes predictions on the test data. Finally, it converts the predicted probabilities to binary class labels, compares the predicted class labels to the true class labels, and calculates the model's accuracy. Please keep in mind that this is just a simple example, in real-world scenarios you should use cross-validation, hyper-parameter

tuning, and many other techniques to improve the model performance.

# GLOSSARY

***A/B testing***

A statistical way of comparing two (or more) techniques—the "A" and the "B"—typically an incumbent against a new rival. A/B testing aims to determine not only which technique performs better but also to understand whether the difference is statistically significant. A/B testing usually considers only two techniques using one measurement, but it can be applied to any finite number of techniques and measurements.

***Accuracy***

The fraction of predictions that a classification model got right. In multi-class classification, accuracy is defined as follows:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Number Of Examples}}$$

In binary classification, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number Of Examples}}$$

See true positive and true negative. Contrast accuracy with precision and recall.

***Activation Function***

A function (for example, ReLU or sigmoid) that takes in the weighted sum of all of the inputs from the previous layer and then

generates and passes an output value (typically nonlinear) to the next layer.

***Backpropagation***

The primary algorithm for performing gradient descent on neural networks. First, the output values of each node are calculated (and cached) in a forward pass. Then, the partial derivative of the error with respect to each parameter is calculated in a backward pass through the graph.

***Binary Classification***

A type of classification task that outputs one of two mutually exclusive classes. For example, a machine learning model that evaluates email messages and outputs either "spam" or "not spam" is a binary classifier.

Contrast with multi-class classification.

***Data Augmentation***

Artificially boosting the range and number of training examples by transforming existing examples to create additional examples. For example, suppose images are one of your features, but your dataset doesn't contain enough image examples for the model to learn useful associations. Ideally, you'd add enough labeled images to your dataset to enable your model to train properly. If that's not possible, data augmentation can rotate, stretch, and reflect each image to produce many variants of the original picture, possibly yielding enough labeled data to enable excellent training.

***Decoder***

In general, any ML system that converts from a processed, dense, or internal representation to a more raw, sparse, or external representation.

Decoders are often a component of a larger model, where they are frequently paired with an encoder.

In sequence-to-sequence tasks, a decoder starts with the internal state generated by the encoder to predict the next sequence.

Refer to Transformer for the definition of a decoder within the Transformer architecture.

***Dimensions***

Overloaded term having any of the following definitions:

- The number of levels of coordinates in a Tensor. For example:
    - A scalar has zero dimensions; for example, ["Hello"].
    - A vector has one dimension; for example, [3, 5, 7, 11].
    - A matrix has two dimensions; for example, [[2, 4, 18], [5, 7, 14]].

You can uniquely specify a particular cell in a one-dimensional vector with one coordinate; you need two coordinates to uniquely specify a particular cell in a two-dimensional matrix.

- The number of entries in a feature vector.
- The number of elements in an embedding layer.

***Discriminator***

A system that determines whether examples are real or fake.

Alternatively, the subsystem within a generative adversarial network that determines whether the examples created by the generator are real or fake.

***Embeddings***

A categorical feature represented as a continuous-valued feature. Typically, an embedding is a translation of a high-dimensional vector into a low-dimensional space. For example, you can represent the words in an English sentence in either of the following two ways:

- As a million-element (high-dimensional) sparse vector in which all elements are integers. Each cell in the vector represents a separate English word; the value in a cell represents the number of times that word appears in a sentence. Since a single English sentence is unlikely to contain more than 50 words, nearly every cell in the vector will contain a 0. The few cells that aren't 0 will contain a low integer (usually 1) representing the number of times that word appeared in the sentence.
- As a several-hundred-element (low-dimensional) dense vector in which each element holds a floating-point value between 0 and 1. This is an embedding.

***Encoder***

In general, any ML system that converts from a raw, sparse, or external representation into a more processed, denser, or more internal representation.

Encoders are often a component of a larger model, where they are frequently paired with a decoder. Some Transformers pair

encoders with decoders, though other Transformers use only the encoder or only the decoder.

Some systems use the encoder's output as the input to a classification or regression network.

In sequence-to-sequence tasks, an encoder takes an input sequence and returns an internal state (a vector). Then, the decoder uses that internal state to predict the next sequence.

Refer to Transformer for the definition of an encoder in the Transformer architecture.

***Epoch***

A full training pass over the entire dataset such that each example has been seen once. Thus, an epoch represents N/batch size training iterations, where N is the total number of examples.

***Feature Extraction***

Overloaded term having either of the following definitions:

- Retrieving intermediate feature representations calculated by an unsupervised or pretrained model (for example, hidden layer values in a neural network) for use in another model as input.
- Synonym for feature engineering.

***Feature Set***

The group of features your machine learning model trains on. For example, postal code, property size, and property condition might comprise a simple feature set for a model that predicts housing prices.

***Feedback Loop***

In machine learning, a situation in which a model's predictions influence the training data for the same model or another model. For example, a model that recommends movies will influence the movies that people see, which will then influence subsequent movie recommendation models.

***Few-Shot Learning***

A machine learning approach, often used for object classification, designed to learn effective classifiers from only a small number of training examples.

***Generalization***

Refers to your model's ability to make correct predictions on new, previously unseen data as opposed to the data used to train the model.

***Heuristic***

A simple and quickly implemented solution to a problem. For example, "With a heuristic, we achieved 86% accuracy. When we switched to a deep neural network, accuracy went up to 98%."

***Hidden Jayer***

A synthetic layer in a neural network between the input layer (that is, the features) and the output layer (the prediction). Hidden layers typically contain an activation function (such as ReLU) for training. A deep neural network contains more than one hidden layer.

***Hyperparameter***

The "knobs" that you tweak during successive runs of training a model. For example, learning rate is a hyperparameter.

***Implicit Bias***

Automatically making an association or assumption based on one's mental models and memories. Implicit bias can affect the following:

- How data is collected and classified.
- How machine learning systems are designed and developed.

For example, when building a classifier to identify wedding photos, an engineer may use the presence of a white dress in a photo as a feature. However, white dresses have been customary only during certain eras and in certain cultures.

***Inference***

In machine learning, often refers to the process of making predictions by applying the trained model to unlabeled examples. In statistics, inference refers to the process of fitting the parameters of a distribution conditioned on some observed data.

***Learning Rate***

A scalar used to train a model via gradient descent. During each iteration, the gradient descent algorithm multiplies the learning rate by the gradient. The resulting product is called the gradient step.

Learning rate is a key hyperparameter.

***Loss***

A measure of how far a model's predictions are from its label. Or, to phrase it more pessimistically, a measure of how bad the model is. To determine this value, a model must define a loss function. For example, linear regression models typically use mean squared error for a loss function, while logistic regression models use Log Loss.

***Model***

The representation of what a machine learning system has learned from the training data.

***Multi-Class Classification***

Classification problems that distinguish among more than two classes. For example, there are approximately 128 species of maple trees, so a model that categorized maple tree species would be multi-class. Conversely, a model that divided emails into only two categories (spam and not spam) would be a binary classification model.

***Pre-Trained Model***

Models or model components (such as embeddings) that have been already been trained. Sometimes, you'll feed pre-trained embeddings into a neural network. Other times, your model will train the embeddings itself rather than rely on the pre-trained embeddings.

***Recurrent Neural Network***

A neural network that is intentionally run multiple times, where parts of each run feed into the next run. Specifically, hidden layers from the previous run provide part of the input to the same

hidden layer in the next run. Recurrent neural networks are particularly useful for evaluating sequences, so that the hidden layers can learn from previous runs of the neural network on earlier parts of the sequence.

For example, the following figure shows a recurrent neural network that runs four times. Notice that the values learned in the hidden layers from the first run become part of the input to the same hidden layers in the second run. Similarly, the values learned in the hidden layer on the second run become part of the input to the same hidden layer in the third run. In this way, the recurrent neural network gradually trains and predicts the meaning of the entire sequence rather than just the meaning of individual words.

***Sequence-to-Sequence Task***

A task that converts an input sequence of tokens to an output sequence of tokens. For example, two popular kinds of sequence-to-sequence tasks are:

- Translators:
  - Sample input sequence: "I love you."
  - Sample output sequence: "Je t'aime."
- Question answering:
  - Sample input sequence: "Do I need my car in New York City?"
  - Sample output sequence: "No. Please keep your car at home."

***Sigmoid Function***

A function that maps logistic or multinomial regression output (log odds) to probabilities, returning a value between 0 and 1. The sigmoid function has the following formula:

$$y = \frac{1}{1 + e^{-\sigma}}$$

where in logistic regression problems is simply:

$$\sigma = b + w_1 x_1 + w_2 x_2 + \ldots w_n x_n$$

In other words, the sigmoid function converts into a probability between 0 and 1.

In some neural networks, the sigmoid function acts as the activation function.

***SoftMax***

A function that provides probabilities for each possible class in a multi-class classification model. The probabilities add up to exactly 1.0. For example, SoftMax might determine that the probability of a particular image being a dog at 0.9, a cat at 0.08, and a horse at 0.02. (Also called full SoftMax.)

***Test Set***

The subset of the dataset that you use to test your model after the model has gone through initial vetting by the validation set.

***Time Series Analysis***

A subfield of machine learning and statistics that analyzes temporal data. Many types of machine learning problems require time series analysis, including classification, clustering, forecasting, and anomaly detection. For example, you could use

time series analysis to forecast the future sales of winter coats by month based on historical sales data.

***Training Set***

The subset of the dataset used to train a model.

***Transfer Learning***

Transferring information from one machine learning task to another. For example, in multi-task learning, a single model solves multiple tasks, such as a deep model that has different output nodes for different tasks. Transfer learning might involve transferring knowledge from the solution of a simpler task to a more complex one, or involve transferring knowledge from a task where there is more data to one where there is less data.

Most machine learning systems solve a single task. Transfer learning is a baby step towards artificial intelligence in which a single program can solve multiple tasks.

***Transformer***

A neural network architecture developed at Google that relies on self-attention mechanisms to transform a sequence of input embeddings into a sequence of output embeddings without relying on convolutions or recurrent neural networks. A Transformer can be viewed as a stack of self-attention layers.

A Transformer can include any of the following:

- an encoder
- a decoder
- both an encoder and decoder

An encoder transforms a sequence of embeddings into a new sequence of the same length. An encoder includes N identical layers, each of which contains two sub-layers. These two sub-layers are applied at each position of the input embedding sequence, transforming each element of the sequence into a new embedding. The first encoder sub-layer aggregates information from across the input sequence. The second encoder sub-layer transforms the aggregated information into an output embedding.

A decoder transforms a sequence of input embeddings into a sequence of output embeddings, possibly with a different length. A decoder also includes N identical layers with three sub-layers, two of which are similar to the encoder sub-layers. The third decoder sub-layer takes the output of the encoder and applies the self-attention mechanism to gather information from it.

***True negative (TN)***

An example in which the model correctly predicted the negative class. For example, the model inferred that a particular email message was not spam, and that email message really was not spam.

***True positive (TP)***

An example in which the model correctly predicted the positive class. For example, the model inferred that a particular email message was spam, and that email message really was spam.

***True positive rate (TPR)***

Synonym for recall. That is:

$$\text{True Positive Rate} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

True positive rate is the y-axis in an ROC curve.

***Validation***

A process used, as part of training, to evaluate the quality of a machine learning model using the validation set. Because the validation set is disjoint from the training set, validation helps ensure that the model's performance generalizes beyond the training set.

***Validation Set***

A subset of the dataset—disjoint from the training set—used in validation.

***Variable Importance***

A set of scores that indicates the relative importance of each feature to the model.

For example, consider a decision tree that estimates house prices. Suppose this decision tree uses three features: size, age, and style. If a set of variable importance for the three features are calculated to be {size=5.8, age=2.5, style=4.7}, then size is more important to the decision tree than age or style.

Different variable importance metrics exist, which can inform ML experts about different aspects of models.

***Weight***

A coefficient for a feature in a linear model, or an edge in a deep network. The goal of training a linear model is to determine the

ideal weight for each feature. If a weight is 0, then its corresponding feature does not contribute to the model.

Source Glossary:

Google, Machine Learning Glossary:
https://developers.google.com/machine-learning/glossary

**Also available from the Author**

# MINDFUL AI

## *Reflections on Artificial Intelligence*

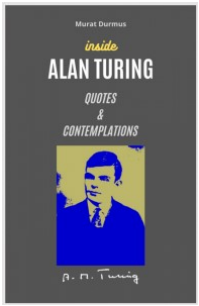
Inspirational Thoughts & Quotes on Artificial Intelligence
(Including 13 illustrations, articles & essays for the fundamental understanding of AI)


Available on Amazon:

**Kindle** -- **Paperback**

Kindle: **(ASIN: B0BKLCKM22)**
Paperback: **(ISBN-13: 979-8360396796)–**

# INSIDE ALAN TURING: QUOTES & CONTEMPLATIONS




Alan Turing is generally considered the father of computer science and artificial intelligence. He was also a theoretical biologist who developed algorithms to explain complex patterns using simple inputs and random fluctuation as a side hobby. Unfortunately, his life tragically ended in suicide in 1954, after he was chemically castrated as punishment (instead of prison) for 'criminal' gay acts.

**"We can only see a short distance ahead, but we can see plenty there that needs to be done." ~ Alan Turing**

Available on Amazon:

**Kindle -- Paperback**

Kindle: **(ASIN: B09K3669BX)**
Paperback: **(ISBN- 979-8751495848)**