



A Journey to Intelligent

Introduction to Machine Learning

Dr. Budditha Hettige
BSc, MPhil, PhD

A Journey to Intelligent

Introduction to Machine Learning

Dr. Buddhitha Hettige

Introduction to Machine Learning

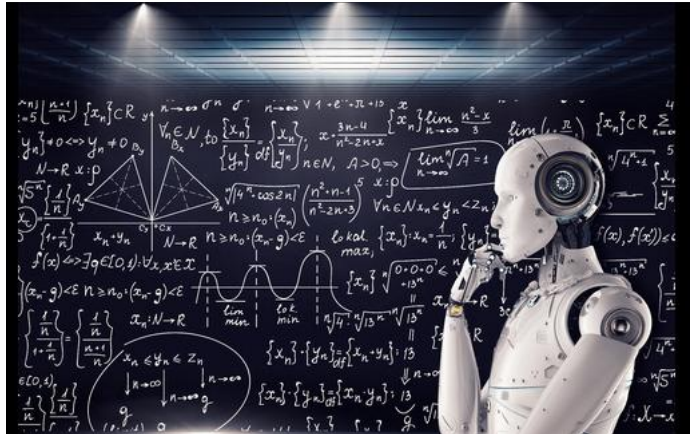
Table of Contents

Machine Learning	1
What is Machine Learning?	1
Machine Learning Approaches:	2
Why Machine Learning?	3
Tradition vs Machine Learning	4
The Traditional Approach	4
Taxonomy of Machine Learning	9
Supervised Learning	10
Typical applications:	12
Algorithms	12
Unsupervised Learning	13
Typical Applications	15
Clustering Algorithms:	15
Dimensionality Reduction Algorithms:	16
Density Estimation Algorithms:	16
Semi-supervised Learning	16
Approaches to semi-supervised learning:	18
Algorithms	18
Reinforcement Learning	20
Algorithms:	23
Main Challenges of Machine Learning	24
Insufficient Quantity of Training Data	24

Data Dimensionality Considerations	26
No representative Training Data	27
Overfitting the Training Data	27
Under fitting the Training Data	29
Testing and Validating	29
Hyperparameter Tuning and Model Selection	30
Data Mismatch	31
ML Deployment Approaches	31
Standard ML Development Stages	32
Cross-industry standard process	32
ML System Development Toolkits	33
ML System Development Toolkits	35
Regression	36
Simple Linear Regression	38
Performance evaluation metrics	42
Problem: Predicting House Prices	43
Multiple Regression	48
Polynomial Regression	52
Example: Quadratic relationship	55
Regularized linear models	58
Hyper parameter Tuning and Model Performance	59
Classification	63
Binary Classification	63
Multiclass Classification	65
Multi-label Classification	68
Imbalance Classification	69

Classification Performance Measures	70
Logistic Regression	76
Support Vector Machine	82
Linear Kernel	86
Polynomial Kernel	87
RBF Kernel.....	89
Decision Tree	91
Ensemble Learning Methods	99
Tree-based Ensemble Learning Algorithms	105
Random Forest.....	105
AdaBoost (Adaptive Boosting)	106
Gradient Boosting Machine (GBM)	107
Extreme Gradient Boosting (XGBoost)	108
Example: Diabetes Patient Prediction	109

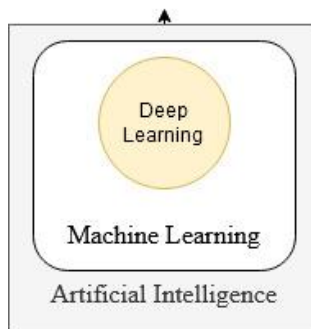
Machine Learning



What is Machine Learning?

Machine learning is a subfield of **Artificial Intelligence (AI)** that focuses on the development of algorithms and statistical models that enable computers to perform tasks without being explicitly programmed. The central idea behind machine learning is to empower machines to learn from data and improve their performance over time (chatGPT).

In other words, Machine Learning is a science (and art) of programming computers so they can learn from data.



Machine Learning Approaches:

1. **Supervised Learning:** In supervised learning, the algorithm is trained on a labeled dataset, where the input data is paired with corresponding output labels. The goal is to learn a mapping from inputs to outputs, allowing the algorithm to make predictions on new, unseen data.
2. **Unsupervised Learning:** Unsupervised learning involves working with unlabeled data. The algorithm explores the inherent structure or patterns within the data without explicit guidance on what to look for. Clustering and dimensionality reduction are common tasks in unsupervised learning.
3. **Reinforcement Learning:** In reinforcement learning, an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions, allowing it to learn optimal strategies over time.
4. **Semi-Supervised Learning:** This approach combines elements of both supervised and unsupervised learning. The algorithm is trained on a dataset with both labeled and unlabeled examples, leveraging the available labeled data while also exploring the underlying structure of the unlabeled data.
5. **Deep Learning:** Deep learning is a subset of machine learning that involves neural networks with multiple layers (deep neural networks). These networks, often referred to as artificial neural networks, can automatically learn hierarchical representations of data, making them particularly effective for tasks like image and speech recognition.

Why Machine Learning?

Machine learning is employed for various reasons across different industries due to its ability to analyze large volumes of data, identify patterns, and make predictions or decisions without explicit programming. Here are some key reasons why machine learning is widely used:

1. **Complex Problem Solving:** Machine learning excels at solving complex problems that are difficult to address with traditional programming methods. It can automatically learn and adapt to changing conditions, making it suitable for tasks where explicit programming might be impractical.
2. **Data Analysis and Pattern Recognition:** Machine learning algorithms can analyze large datasets to identify patterns, trends, and insights that may not be immediately apparent to humans. This is particularly useful for extracting valuable information from vast amounts of data.
3. **Automation:** Machine learning enables automation of tasks that would otherwise require human intervention. This includes automating repetitive and time-consuming processes, allowing human resources to focus on more strategic and creative aspects of their work.
4. **Prediction and Forecasting:** Machine learning models are capable of making predictions based on historical data. This is valuable in fields such as finance, weather forecasting, stock market analysis, and demand forecasting, where accurate predictions are essential for decision-making.
5. **Personalization and Recommendations:** Many online services use machine learning to provide personalized recommendations based on user behavior and preferences. This is evident in platforms like Netflix, Amazon, and social media,

where algorithms analyze user data to suggest relevant content or products.

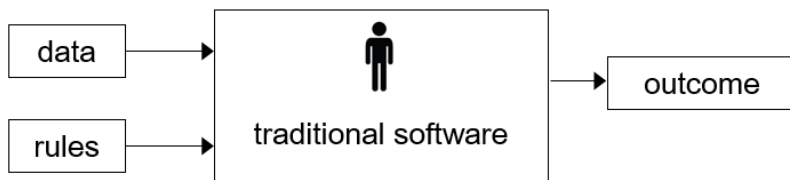
6. **Natural Language Processing (NLP):** Machine learning plays a crucial role in NLP, allowing computers to understand, interpret, and generate human language. Applications include language translation, chatbots, sentiment analysis, and voice recognition.
7. **Image and Speech Recognition:** Machine learning is widely used in image and speech recognition applications. This is evident in facial recognition systems, medical image analysis, voice assistants like Siri and Alexa, and security systems.


Tradition vs Machine Learning

To take clear idea about ML lets looks on Tradition programming way and ML method.

The Traditional Approach

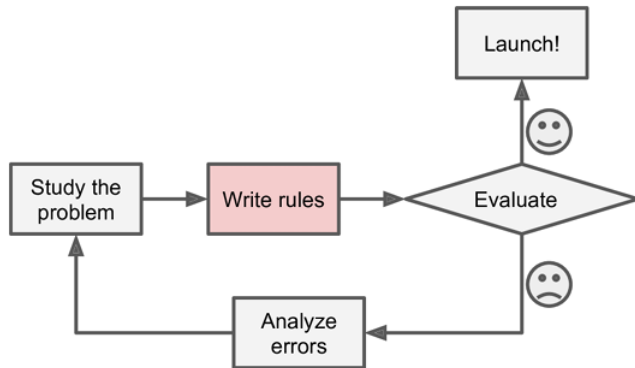
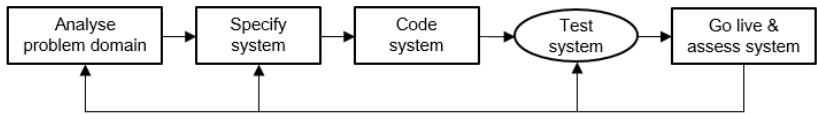
In traditional software, we discover rules/patterns/model manually and encode them with a programming language.



 Encode

For that you need to think on algorithms/ methods. Then we need to write a program to by looking on inputs.

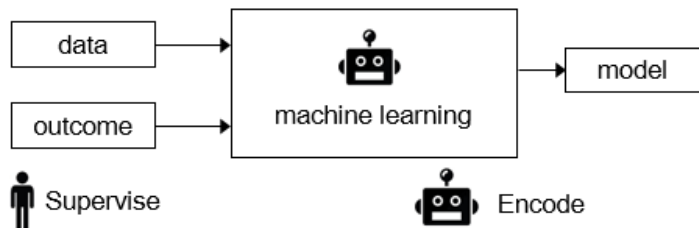
The following diagram show to way how we develop this kind of programs.



Now we can see how machine learning method works.

Machine Learning Approach

A Machine Learning system discovers rules/patterns/model automatically by learning from examples. After training it produces a model that “knows” these patterns, but we still need to supervise it to make sure the model is correct.



In here you no need to think on algorithms/ methods. System has an ability to find the suitable model.

Example:

Create a simple python program to convert Celsius Temperature into Fahrenheit;

```
def celsius_to_fahrenheit(celsius):  
    fahrenheit = (celsius * 9/5) + 32  
    return fahrenheit  
  
# Example usage  
celsius_value = 37  
fahrenheit_value = celsius_to_fahrenheit(celsius_value)  
print(f"{celsius_value}°C is equal to {fahrenheit_value}°F")
```

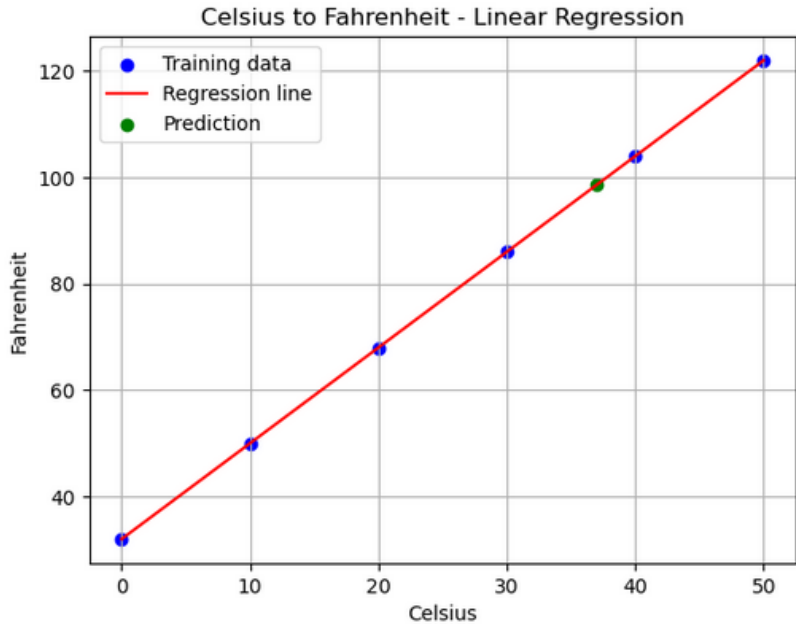
37°C is equal to 98.6°F

The following diagram show to say how we develop this kind of programs

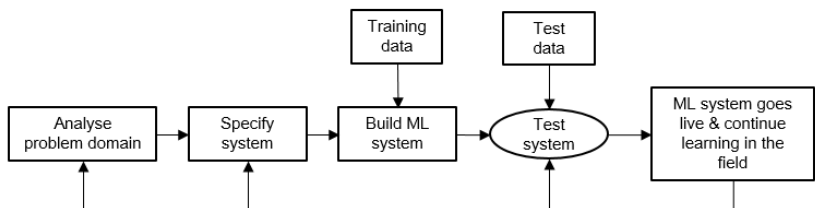
Now same can be done through the ML methods

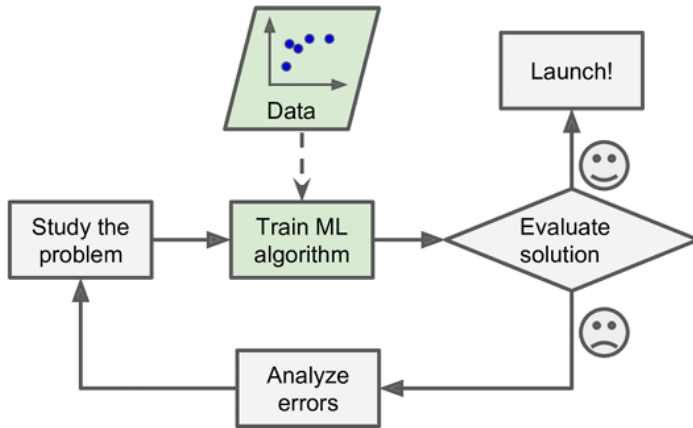
```
import numpy as np  
from sklearn.linear_model import LinearRegression  
import matplotlib.pyplot as plt  
  
# Sample data: Celsius and corresponding Fahrenheit  
celsius = np.array([[0], [10], [20], [30], [40], [50]])  
fahrenheit = np.array([[32], [50], [68], [86], [104], [122]])  
  
# Create and train the model  
model = LinearRegression()  
model.fit(celsius, fahrenheit)  
  
# Predict a new value  
new_celsius = 37  
predicted_fahrenheit = model.predict([[new_celsius]])  
print(f"Predicted: {new_celsius}°C is approximately {predicted_fahrenheit[0][0]:.2f}°F")  
  
# Optional: Plotting the data and prediction  
plt.scatter(celsius, fahrenheit, color='blue', label='Training data')  
plt.plot(celsius, model.predict(celsius), color='red', label='Regression line')  
plt.scatter([new_celsius], predicted_fahrenheit, color='green', label='Prediction')  
plt.xlabel("Celsius")  
plt.ylabel("Fahrenheit")  
plt.legend()  
plt.title("Celsius to Fahrenheit - Linear Regression")  
plt.grid(True)  
plt.show()
```

Predicted: 37°C is approximately 98.60°F

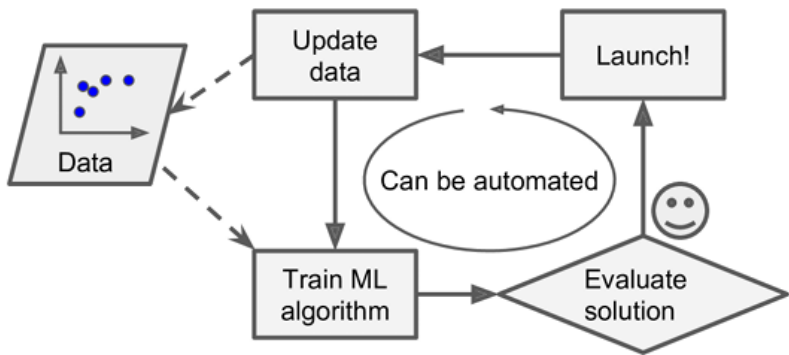


In this example we take sample data from Celsius and Fahrenheit but we don't have an idea on how we take Celsius to Fahrenheit it takes by the ML system.

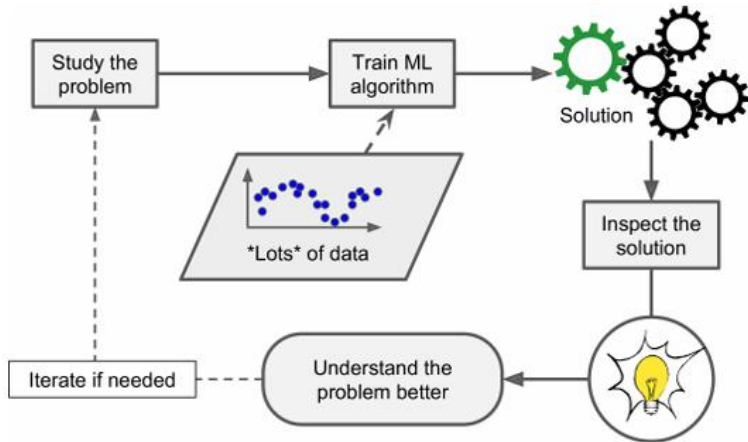




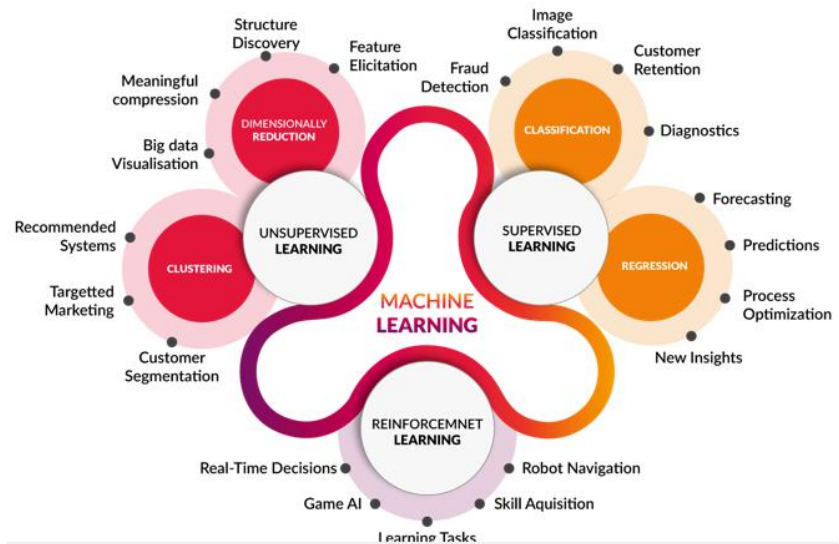
One of the most important feature on this method allows to automatically Adapting to Change



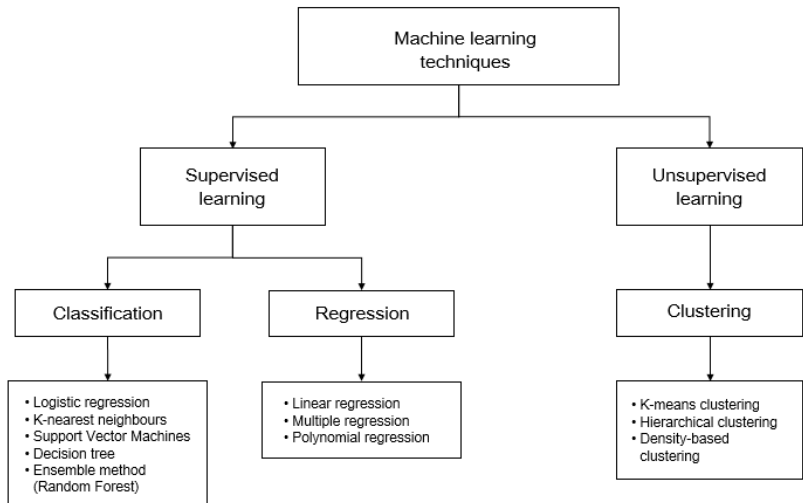
Also Machine Learning can help Humans Learning



Taxonomy of Machine Learning



Summary of the machine learning techniques.



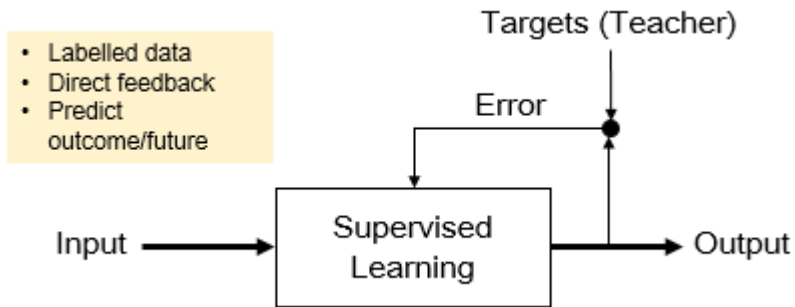
Now we take more details on different ML Methods

Supervised Learning

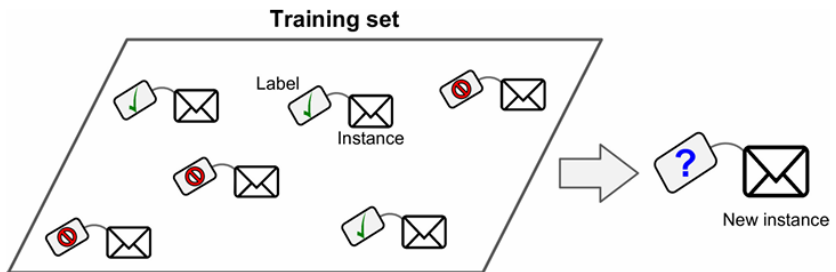
Supervised learning requires labelled training and test datasets that have known independent X-variables (features) and known output Y-variable values.

ML algorithms then determine how the X and Y variables are best related, thereby creating a trained prediction model, which can then be applied against real non-labelled datasets to predict Y-variable values from the inputs.

The error is the direct feedback to model in order to predict outcome/future in order to decrease the error.

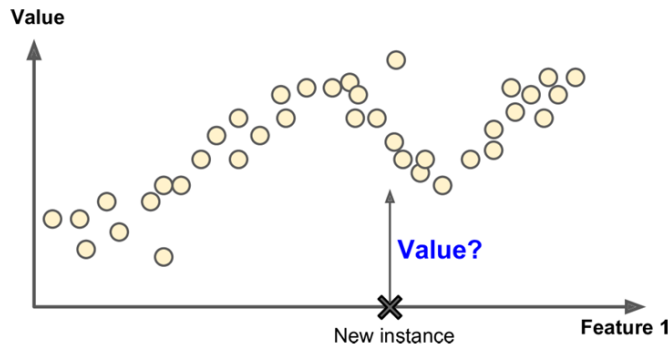


In supervised learning, the training data you feed to the algorithm includes the desired solutions, called labels



A typical supervised learning task is **classification**. The spam filter is a good example of this: it is trained with many example emails along with their class (spam or ham), and it must learn how to classify new emails

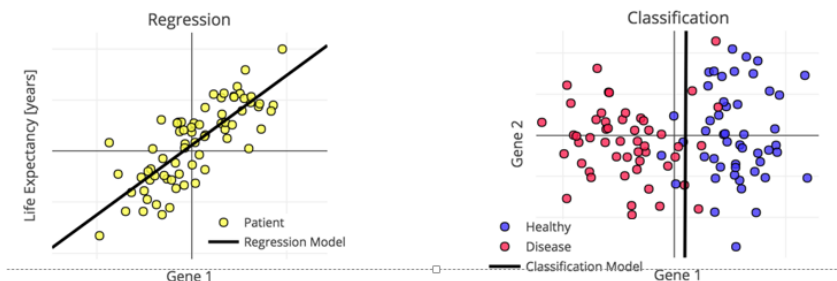
Another typical task is to **predict a target numeric value**, such as the price of a car, given a set of features (mileage, age, brand, etc.) called predictors. This sort of task is called regression (Figure 1-6).¹ To train the system, you need to give it many examples of cars, including both their predictors and their labels.



Typical applications:

Regression: process of predicting a continuous, numerical value for input data sample e.g. house price, temperature forecasting.

Classification: process of assigning category to input data sample e.g. predicting whether a person is ill or not, detecting fraudulent transactions, face classifier.



Algorithms

Here are some of the most important supervised learning algorithms

1. **Linear Regression:** A simple algorithm used for regression tasks, where the goal is to predict a continuous numeric value based on input features.

2. **Logistic Regression:** Despite its name, logistic regression is used for binary classification problems, predicting whether an instance belongs to one of two classes.
3. **Support Vector Machines (SVM):** Effective for both classification and regression tasks, SVM aims to find a hyperplane that best separates data points of different classes in a high-dimensional space.
4. **Decision Trees:** Trees that recursively split data based on features, making decisions at each node to classify instances or predict values.
5. **Random Forest:** An ensemble learning method that builds multiple decision trees and combines their predictions, often providing improved performance and robustness.
6. **K-Nearest Neighbors (KNN):** A simple algorithm that classifies instances based on the majority class of their nearest neighbors in the feature space.
7. **Naïve Bayes:** A probabilistic algorithm based on Bayes' theorem, commonly used for classification tasks, particularly in natural language processing and spam filtering.

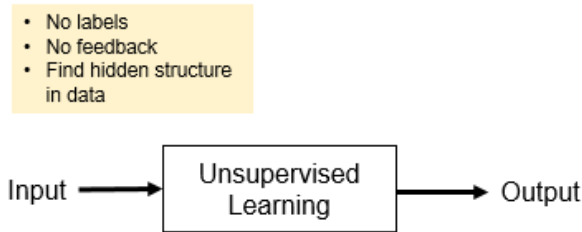
Unsupervised Learning

In unsupervised learning, as you might guess, the training data is **unlabeled**. The system tries to learn without a teacher

Unsupervised learning is a type of machine learning where the algorithm is given data without explicit instructions on what to do with it. Unlike supervised learning, there are no labeled output variables to guide the learning process. Instead, the algorithm explores the data's inherent structure or patterns to make sense of it.

Unsupervised learning system tries to identify patterns within non-labelled data that allows a model to be constructed for predicting Y-variable values given a known set of independent X-

variables (features). In other words, unsupervised learning system attempt to group similar data points within the dataset based on previously set criteria. Each grouping points to a Y-value indication. The grouping or clustering rules are used to create a prediction model.



The primary goals of unsupervised learning are often:

Clustering: Grouping similar data points together based on certain features or characteristics. Clustering algorithms aim to find natural groupings in the data without any predefined categories.

Example: K-Means clustering algorithm, hierarchical clustering.

Dimensionality Reduction: Reducing the number of features in a dataset while preserving its essential information. This can be useful for simplifying models, visualizing data, and eliminating noise.

Example: Principal Component Analysis (PCA), t-Distributed Stochastic Neighbor Embedding (t-SNE).

Density Estimation: Estimating the probability density function of the underlying data distribution. This can be helpful in anomaly detection or understanding the distribution of data points in a continuous space.

Example: Gaussian Mixture Models (GMM), Kernel Density Estimation (KDE).

Typical Applications

- **Customer Segmentation:** Identifying groups of customers with similar behaviors or characteristics for targeted marketing strategies.
- **Anomaly Detection:** Detecting unusual patterns or outliers in data, which can indicate fraud, defects, or other abnormal events.
- **Topic Modeling:** Extracting themes or topics from a collection of documents without prior knowledge of the categories.
- **Data Compression:** Reducing the dimensionality of data for more efficient storage and processing.
- **Feature Learning:** Automatically learning useful representations or features from raw data.
- **Image and Speech Recognition:** Unsupervised learning can be used for pretraining models or extracting features before fine-tuning in a supervised manner.

Clustering Algorithms:

- **K-Means:** Divides data into 'k' clusters based on similarity, with each cluster represented by its centroid.
- **Hierarchical Clustering:** Builds a tree of clusters, where the leaf nodes represent individual data points, and internal nodes represent merged clusters.
- **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise): Groups together data points that are close to each other and separates regions with lower point density.
- **Mean Shift:** Identifies clusters by shifting towards areas of higher data point density.

- **Agglomerative Clustering:** Starts with individual data points and gradually merges them into clusters based on certain criteria.

Dimensionality Reduction Algorithms:

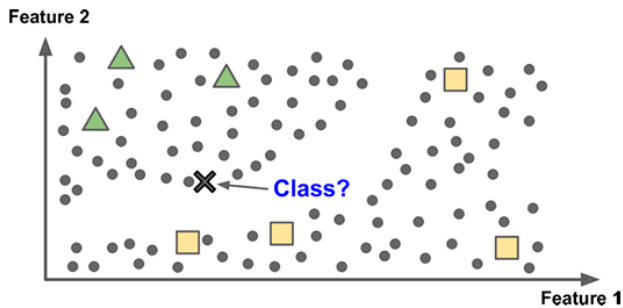
- **Principal Component Analysis (PCA):** Reduces the dimensionality of data by transforming it into a set of orthogonal (uncorrelated) principal components.
- **t-Distributed Stochastic Neighbor Embedding (t-SNE):** Emphasizes preserving local relationships between data points in a reduced-dimensional space, often used for visualization.
- **Autoencoders:** Neural network-based models that learn efficient representations of input data by compressing it into a lower-dimensional space and then reconstructing the input.

Density Estimation Algorithms:

- **Gaussian Mixture Models (GMM):** Models the distribution of data as a mixture of multiple Gaussian distributions.
- **Kernel Density Estimation (KDE):** Estimates the probability density function of a continuous random variable based on its sample data.

Semi-supervised Learning

Some algorithms can deal with partially labeled training data, usually a lot of unlabeled data and a little bit of labeled data. This is called semi supervised learning.



Semisupervised learning algorithms are combinations of unsupervised and supervised algorithms

Semi-supervised learning combines elements of both approaches, leveraging the benefits of labeled data while also capitalizing on the larger pool of unlabeled data. The main scenarios where semi-supervised learning is useful include:

1. **Limited Labeled Data:** When obtaining labeled data is expensive, time-consuming, or challenging, semi-supervised learning allows the model to generalize better by using a smaller set of labeled examples and a larger set of unlabeled examples.
2. **Data Annotation Challenges:** In many real-world applications, labeling data can be a bottleneck due to the need for domain expertise. Semi-supervised learning can be particularly beneficial when it's difficult to obtain accurate labels for a large dataset.
3. **Continuous Learning:** In dynamic environments where the distribution of data is continually changing, semi-supervised learning enables models to adapt and learn from new, unlabeled data over time without requiring constant re-labeling.

Approaches to semi-supervised learning:

1. **Self-training:** The model is initially trained on the small labeled dataset. It then makes predictions on the unlabeled data, and high-confidence predictions are added to the training set as labeled examples for subsequent training iterations.
2. **Co-training:** The model is trained on different subsets of features or views of the data. Each view may have a small amount of labeled data, and the model is expected to agree on predictions across these views.
3. **Multi-view Learning:** Similar to co-training, but different models are trained on different views of the data. Each model might have access to a different set of labeled data or features.
4. **Generative Models:** Unlabeled data is used to train a generative model (e.g., autoencoder, variational autoencoder), which can then be used to generate additional labeled examples for training a supervised model.

Semi-supervised learning methods aim to strike a balance between utilizing labeled information for guidance and leveraging the vast amount of unlabeled data to improve model performance and generalization. The effectiveness of semi-supervised learning depends on the specific characteristics of the dataset and the problem being addressed.

Algorithms

Here are some notable semi-supervised learning algorithms:

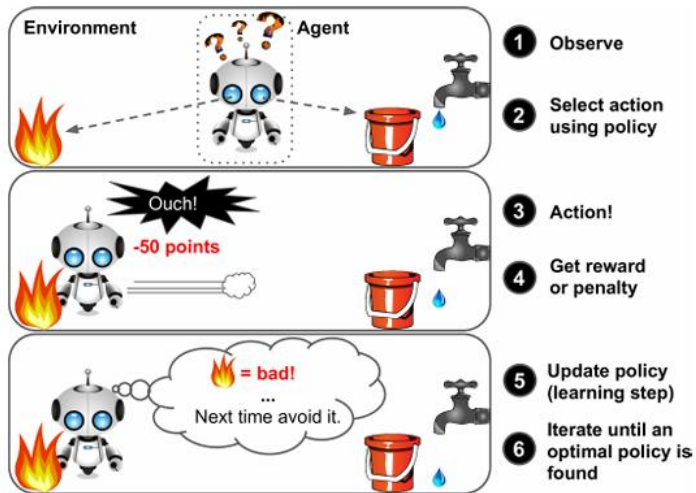
1. **Self-training:** A simple and commonly used approach where a model is initially trained on a small labeled dataset. It then makes predictions on unlabeled data, and

the high-confidence predictions are added to the labeled dataset for subsequent training iterations.

2. **Co-training:** This approach involves training multiple models on different subsets of features or views of the data. Each model may have access to a different set of labeled data or features, and the models iteratively share information to improve performance.
3. **Multi-view Learning:** Similar to co-training, multi-view learning involves training different models on different views or representations of the data. Each view might correspond to a subset of features or a different representation of the input data.
4. **Generative Models:** Generative models, such as autoencoders and variational autoencoders, can be trained on unlabeled data to learn a generative distribution. The generated data can then be used to augment the labeled dataset for training a supervised model.
5. **Semi-Supervised Support Vector Machines (S3VM):** An extension of traditional Support Vector Machines (SVM) to semi-supervised scenarios, where the SVM is trained on a combination of labeled and unlabeled data.
6. **Label Propagation:** In label propagation methods, labels from a small set of labeled instances are propagated to unlabeled instances based on their similarity in the feature space.
7. **Tri-Training:** An extension of self-training where three models are trained on three different subsets of features. Each model then labels the unlabeled data, and instances on which all three models agree are added to the labeled dataset.
8. **Mean Teacher:** A type of consistency regularization method where a student model is trained to match the predictions of a moving average (mean) of the teacher model on both labeled and unlabeled data.

Reinforcement Learning

Reinforcement Learning is a very different beast. The learning system, called an agent in this context, can observe the environment, select and perform actions, and get rewards in return (or penalties in the form of negative rewards, as in Figure). It must then learn by itself what is the best strategy, called a policy, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation



Reinforcement Learning (RL) is a type of machine learning paradigm where an agent learns to make decisions by interacting with an environment. The agent learns through trial and error, receiving feedback in the form of rewards or penalties based on the actions it takes. The primary goal of reinforcement learning is to find an optimal strategy, called a policy, that maximizes the cumulative reward over time.

Components of Reinforcement Learning

1. **Agent:** The entity that makes decisions and takes actions in the environment. The agent's objective is to learn a policy that guides its actions to maximize the cumulative reward.
2. **Environment:** The external system with which the agent interacts. The environment responds to the agent's actions, providing feedback in the form of rewards or penalties, and transitioning to new states.
3. **State:** A representation of the current situation or configuration of the environment. The state provides information about the context in which the agent is making decisions.
4. **Action:** The set of possible moves or decisions that the agent can take in a given state. Actions lead to transitions to new states and may result in receiving rewards or penalties.
5. **Reward:** A numerical value that the environment provides to the agent after each action, indicating the immediate benefit or cost associated with the action. The agent's goal is to maximize the cumulative reward over time.
6. **Policy:** The strategy or mapping from states to actions that the agent learns. The policy guides the agent's decision-making process, determining which action to take in each state.

RL process typically involves the following steps:

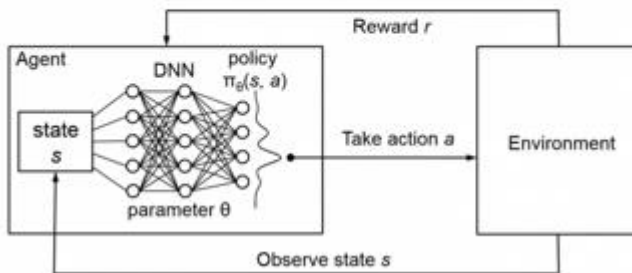
1. **Observation:** The agent observes the current state of the environment.
2. **Decision:** The agent selects an action based on its current policy.

3. **Action:** The selected action is taken, leading to a transition to a new state.
4. **Reward:** The environment provides a reward or penalty based on the action taken and the resulting state.
5. **Learning:** The agent updates its policy based on the observed rewards, aiming to improve its decision-making strategy over time.

Reinforcement learning has been successfully applied to various domains, including robotics, game playing (e.g., AlphaGo), autonomous systems, finance, and healthcare. Reinforcement learning involves an agent observing the state based on the information received from the environment, which is stored and then used for choosing the right action.

Training of an agent is a process of trial and error in various situations and learning from receiving a reward that can be either positive or negative. The reward is a feedback that can be used by an agent to update its parameters.

- Decision process
- Reward system
- Learn series of actions



Algorithms:

1. **Q-Learning:** A classic model-free algorithm that learns a Q-value for each state-action pair, representing the expected cumulative reward when taking a specific action in a particular state.
2. **Deep Q Networks (DQN):** Extends Q-learning by using deep neural networks to approximate the Q-values. DQN has been successful in solving complex tasks, such as playing Atari 2600 games.
3. **Policy Gradient Methods:** A family of algorithms that directly optimize the policy of the agent. Examples include:
 - **REINFORCE:** Uses the policy gradient theorem to update the policy parameters based on the expected cumulative reward.
 - **Proximal Policy Optimization (PPO):** A popular policy optimization algorithm that ensures stable and efficient updates to the policy.
 - **Trust Region Policy Optimization (TRPO):** Another policy optimization algorithm that uses a trust region to limit the size of policy updates.
4. **Actor-Critic Methods:** Combines value-based (critic) and policy-based (actor) approaches. Examples include:
 - **Advantage Actor-Critic (A2C):** Updates both the policy and value function in an asynchronous manner, typically using multiple actors.
 - **Asynchronous Advantage Actor-Critic (A3C):** An asynchronous version of A2C that scales better to distributed computing environments.
5. **Deep Deterministic Policy Gradients (DDPG):** An off-policy actor-critic algorithm that extends ideas from DQN to continuous action spaces. It's commonly used in robotic control tasks.
6. **Twin Delayed DDPG (TD3):** An improvement upon DDPG that introduces several modifications to stabilize

training, such as using twin critics and delayed policy updates.

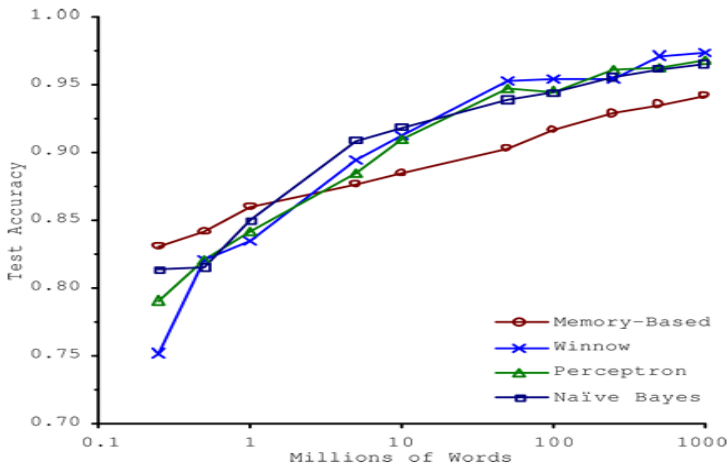
7. **Soft Actor-Critic (SAC):** A model-free, off-policy algorithm that incorporates entropy regularization to encourage exploration.
8. **Hindsight Experience Replay (HER):** A modification to reinforcement learning algorithms that use off-policy learning to handle sparse rewards by relabeling unsuccessful trajectories.
9. **Monte Carlo Tree Search (MCTS):** A planning algorithm used in combination with reinforcement learning for decision-making in games and other sequential decision processes. AlphaGo used an MCTS variant for decision-making.
10. **Multi-Agent Deep Deterministic Policy Gradients (MADDPG):** An extension of DDPG designed for multi-agent environments.

Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the two things that can go wrong are “**bad algorithm**” and “**bad data**.” Let’s start with examples of bad data.

Insufficient Quantity of Training Data

Machine Learning is not quite there yet; it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples



In real life scenarios, data that is obtained for use in developing and working with ML systems is often imperfect. This may manifest as missing, duplicate or incorrectly entered data values.

At the core of modern ML-based systems which depend on data to derive their predictive power. Because of this, all ML projects are dependent on high data quality.

There are numerous data quality issues that can threaten to derail ML systems the following issues need to be considered and prevented before issues arise:

- Inaccurate, incomplete and improperly labelled data
- Having too much data
- Having too little data
- Biased data
- Unbalanced data
- Inconsistent data

For most ML applications therefore, the first steps after injecting data is to carry out data cleansing (also termed data scrubbing or data cleaning).

Data Dimensionality Considerations

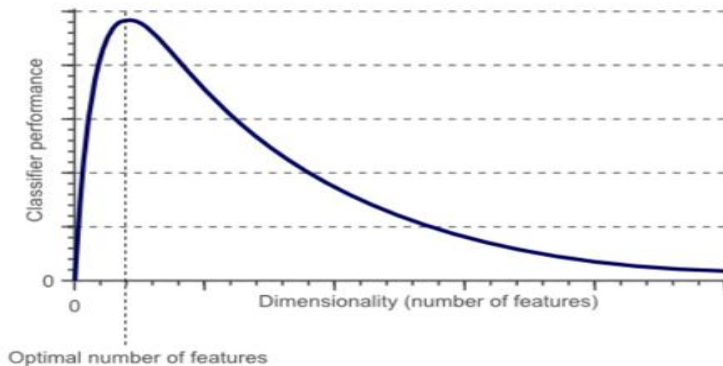
Dimensionality in statistics refers to how many attributes a dataset has. In other words, it refers to the number of features (X-input variables) that are deemed relevant to creating the ML model.

For example, healthcare data is notorious for having vast amounts of variables (e.g. blood pressure, weight, cholesterol level).

In practice, this is difficult to do, in part because many variables are inter-related (like weight and blood pressure).

With high Dimensional data, the number of features can exceed the number of observations. Moreover, working with **too many dimensions** increases the likelihood of **overfitting** the model, which generally reduces the predictive performance with real/live data in the field.

For example, variables are inter-related e.g. weight and blood pressure. A known term the Curse of Dimensionality which is exemplified by the graph.

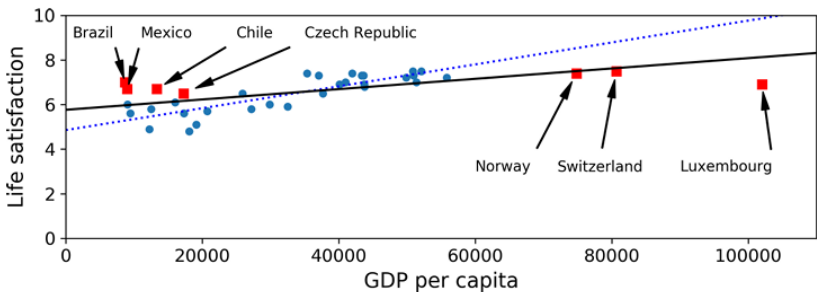


As the dimensionality increases, the ML classifier's performance increases until the optimal number of features is reached.

Further increasing the dimensionality without increasing the number of training samples results in a decrease in classifier performance.

No representative Training Data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning. For example, the set of countries we used earlier for training the linear model was not perfectly representative; a few countries were missing. Figure shows what the data looks like when you add the missing countries.



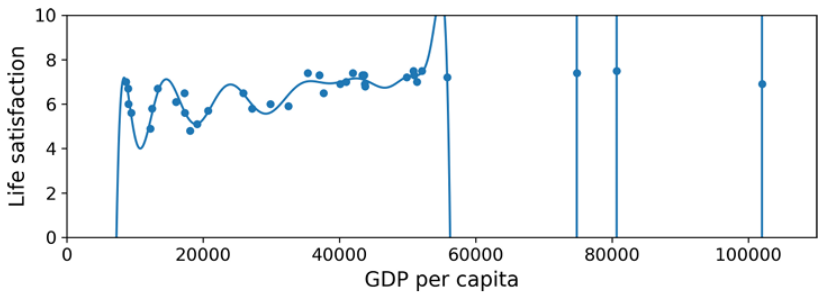
Also we need to think;

- Sampling Bias
- Poor-Quality Data
- Irrelevant Features

Overfitting the Training Data

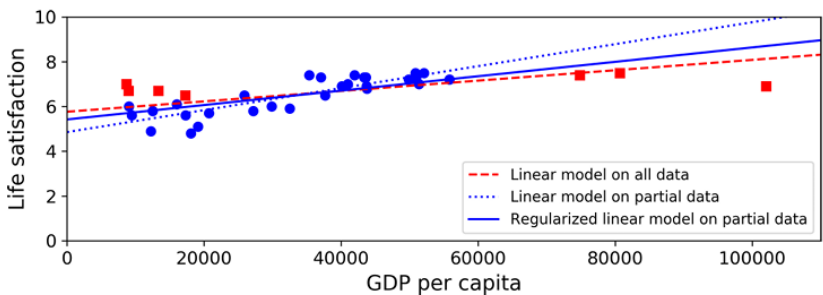
Figure shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even

though it performs much better on the training data than the simple linear model, would you really trust its predictions?



Regularization reduces the risk of overfitting

The amount of regularization to apply during learning can be controlled by a **hyper parameter**. A hyperparameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. If you set the regularization hyper parameter to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not overfit the training data, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a Machine Learning system.



Under fitting the Training Data

As you might guess, under fitting is the opposite of overfitting: it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples. The main options to fix this problem are:

- Selecting a more powerful model, with more parameters
- Feeding better features to the learning algorithm (feature engineering)
- Reducing the constraints on the model (e.g., reducing the regularization hyper parameter)

Testing and Validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea. A better option is to split your data into two sets: the training set and the test set. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the generalization error (or out-of sample error), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.

It is common to **use 80%** of the data for training and hold out **20% for testing**. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances: that's probably more than enough to get a good estimate of the generalization error.

Hyperparameter Tuning and Model Selection

So evaluating a model is simple enough: just use a test set. Now suppose you are hesitating between two models (say a linear model and a polynomial model): how can you decide? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better, but you want to apply some regularization to avoid overfitting. The question is: how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error, say just 5% error. So you launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model for that particular set. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is called holdout validation: you simply hold out part of the training set to evaluate several candidate models and select the best one.

Data Mismatch

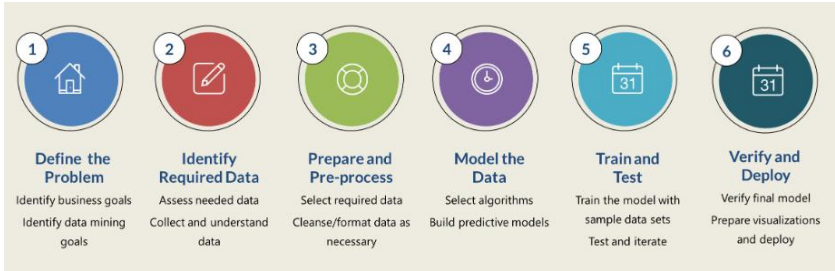
In some cases, it is easy to get a large amount of data for training, but it is not perfectly representative of the data that will be used in production. For example, suppose you want to create a mobile app to take pictures of flowers and automatically determine their species. You can easily download millions of pictures of flowers on the web, but they won't be perfectly representative of the pictures that will actually be taken using the app on a mobile device. Perhaps you only have 10,000 representative pictures.

ML Deployment Approaches

ML has been applied for solving problems in areas such as:

- Financial analyses, consumer credit scoring and automated trading systems
- Energy and pollution modelling/forecasting
- Market segmentation and analysis
- Feature/facial recognition, image processing and computer vision
- Medical diagnoses, cancer detection, drug synthesis and DNA sequencing
- Self-driving vehicles
- Automated manufacturing systems
- Natural language processing

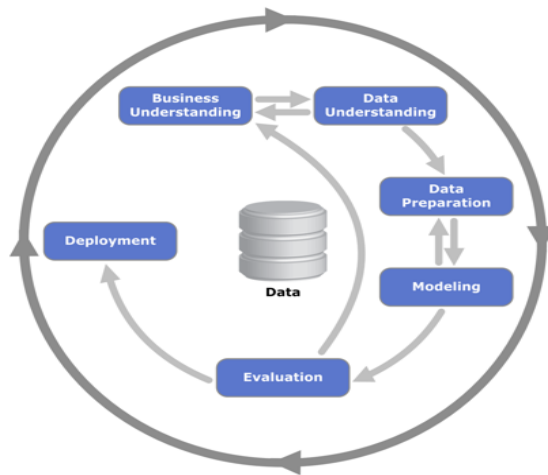
Standard ML Development Stages



Cross-industry standard process

Cross-industry standard process for data mining, known as CRISP-DM, is an open standard process model that describes common approaches used by ML experts.

CRISP-DM breaks the process of data mining into six major phases. It is the most widely-used model for deploying ML system.



ML System Development Toolkits

In this module, we will be using some essential software packages to develop ML systems including:

- Loading data from a variety of sources
- Manipulating data, typically as frames
- Performing statistical analysis
- Visualization of data
- Implement ML model
- Model evaluation

Some of the existing tools and languages;

Python is the programming language framework, which is commonly used for ML, AI, data science etc.

Jupyter-Notebook is the integrated development environment (IDE) for Python offered as part of the Anaconda software. It provides an interactive Python command shell which is based on shell, web browser, and the application interface), with graphical integration, customisable commands, rich history (in the JSON format), and computational parallelism for an enhanced performance.

NumPy is the true analytical workhorse of the Python language. It provides the user with multidimensional arrays, along with a large set of functions to operate a multiplicity of mathematical operations on these arrays. Arrays are blocks of data that are arranged along multiple dimensions, which implement mathematical vectors and matrices. Characterized by optimal memory allocation, arrays are useful—not just for storing data, but also for fast matrix operations (vectorization), which are

indispensable when you wish to solve adhoc data science problems.

The pandas package deals with everything that NumPy cannot do. Thanks to its specific data structures, namely DataFrames and Series, pandas allows you to handle complex tables of data of different types (which is something that NumPy's arrays cannot do) and time series. You will be able to easily and smoothly load data from a variety of sources. You can then slice, dice, handle missing elements, add, rename, aggregate, reshape, and finally visualize your data at will.

Matplotlib is a library that contains all the building blocks that are required to create quality plots from arrays and to visualize them interactively. It offers a variety of graph types e.g. bar-charts, histograms, X-Y line graphs, pie charts, scatter plots etc., which can be generated and customized using a small set of commands.

Seaborn is a high-level visualisation package based on matplotlib and integrated with pandas data structures (such as Series and DataFrames) capable to produce informative and beautiful statistical visualizations.

Scikit-learn is the core of data science operations in Python. It offers all that you may need in terms of data pre-processing, supervised and unsupervised learning, model selection, validation, and error metrics.

Anaconda creates a virtual environment within an OS to allow Python workflow

ML System Development Toolkits

A deep learning framework is an interface, library or a tool which allows us to build deep learning models more easily and quickly, without getting into the details of underlying algorithms. They provide a clear and concise way for defining models using a collection of pre-built and optimised components. Below are some of the key features of a good deep learning framework:

- Easy to understand and code
- Automatically compute gradients i.e. backpropagation
- Optimized for performance such as fast GPU/CPU implementation of matrix multiplication, convolutions and backpropagation
- Process parallelism to reduce computations
- Good community supports and contributions e.g. open source codes, models, pre-trained model etc.



Now take some summary on common ML methods

Regression

Regression is one of the main types of supervised learning in ML. Regression analysis is a form of predictive modelling technique which investigates the relationship between a dependent (target) variable and independent variable(s) (predictor). The benefits of using regression analysis are as follows:

- It indicates the significant relationships between dependent variable and independent variable.
- It indicates the strength of impact of multiple independent variables on a dependent variable.
- These benefits help data analysts / data scientists to eliminate and evaluate the best set of variables to be used for building predictive models.

This technique is used for forecasting, time series modelling and finding the causal effect relationship between the variables on a continuous scale. Typical regression problems include the following:

- **Predicting house prices** based on location, square footage, number of bedrooms, and bathrooms
- **Estimating power consumption** based on information about a system's processes and memory
- **Forecasting** demand in retail
- **Predicting** stock prices
- Forecasting weather

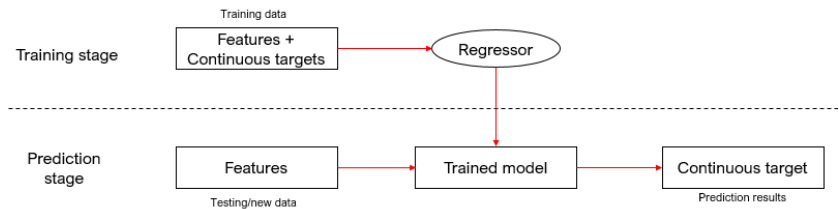
In regression, the training set contains observations (also called features) and their associated continuous target values.

The first columns are known as the features , or also commonly referred to as the independent variables .

The last column is known as the label (or target), or commonly called the dependent variable (or dependent variables if there is more than one label).

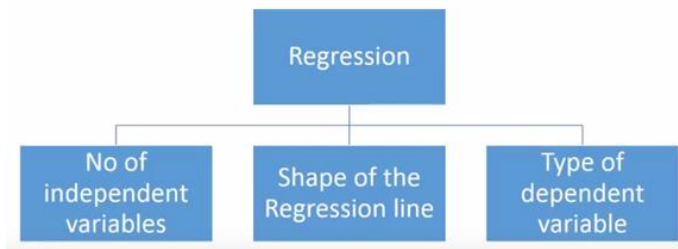
The regression process has two phases:

1. The first phase is **exploring the relationships between the observations and the targets**. This is the training phase.
2. The second phase is using the patterns from the first phase to **generate the target for a future observation**. This is the prediction phase.



There are various kinds of regression techniques available to make predictions.

These techniques are mostly driven by three metrics (number of independent variables, type of dependent variables and shape of regression line).



Simple Linear Regression: Linear relationship between one independent variable and one dependent variable

Multiple Regression: Linear relationships between two or more independent variables and one dependent variable.

Polynomial Regression: Modelling the relationship between one independent variable and one dependent variable using an nth degree polynomial function.

Polynomial Multiple Regression: Modelling the relationship between two or more independent variables and one dependent variable using an nth degree polynomial function.

Simple Linear Regression

The goal of simple (univariate) linear regression is a model the relationship between a single feature (explanatory variable, x) and a continuous-valued target (response variable, y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1x \quad \text{aka. } y = wx + b$$

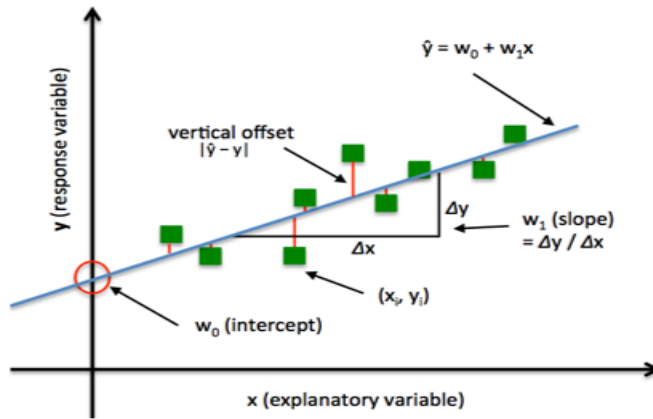
where

y = dependent variable

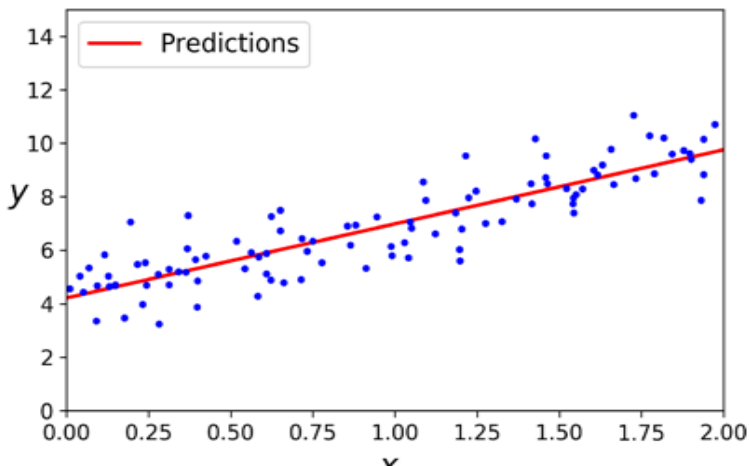
x = explanatory variable

w_0 = y-axis intercept (constant term)

w_1 = weight coefficients of the explanatory variable



- Linear regression tries to fit as many of the data points as possible with a straight line in two-dimensional space.
- It explores the linear relationship between observations and targets, and the relationship is represented in a linear equation or weighted sum function.
- This best-fit line is also called the regression line, and the vertical lines from the regression line to the training examples are the so-called offsets or residuals—the errors of our prediction.



- The main purpose of the best-fit line is that the predicted values y should be closer to the actual or the observed values y .
- In other words, we tend to minimise the difference between the values predicted by us and the observed values, and which is actually termed as error (aka. residual, offset).
- The errors are indicated by the vertical lines showing the difference between the predicted and actual value.

Mean Absolute Error (MAE)

Calculates the average of the absolute difference between the actual and predicted values in the dataset. It measures the average of the residuals (erros) in the dataset.

Absolute difference means that result has no negative sign.

$$\text{MAE} = \frac{1}{m} \sum_{i=1}^m |y^{(i)} - \hat{y}^{(i)}|$$

Mean Squared Error (MSE)

Calculates the average of the squared difference between the original and predicted values in the data set. It measures the variance of the residuals.

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

Root Mean Squared Error (RMSE)

It is the square root of MSE. It measures the standard deviation of residuals.

$$\text{RMSE} = \sqrt{\frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2}$$

R squared (R^2)

The coefficient of determination or R-squared represents the proportion of the variance in the dependent variable of the linear regression model. In other words, it indicates how close the regression line. R-squared score lies between 0 and 1 where 0 indicates that this model does not fit the given data, and 1 indicates that the model fits perfectly to the data provided.

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}}$$

$$\text{RSS} = \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})^2$$

$$\text{TSS} = \sum_{i=1}^m (y^{(i)} - \bar{y}^{(i)})^2$$

RSS=Residual sum of squares

TSS=Total sum of squares

$y^{(i)}$ = target value in a test sample

$y^{(i)}$ = target value in a test sample

$\hat{y}^{(i)}$ = predicted value of a test sample

$\bar{y}^{(i)}$ = mean value of a test sample

Adjusted R squared (R^2)

A modified version of R square, and it is adjusted for the number of independent variables in the model, and it will always be less than or equal to R^2 . In the formula below n is the number of observations in the data and k is the number of the independent variables in the data.

$$R^2 = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

Performance evaluation metrics

MSE is a differentiable function that makes it easy to perform mathematical operations in comparison to a non-differentiable function like MAE.

MSE (sometime RMSE) it is widely used as a default metric for calculating Cost/Loss Function despite being harder to interpret than MAE.

MSE is sensitive to “noisy” dataset i.e. contains outliers, or unexpected values (too high or too low values), hence it is a useful indicator of the model's predicting abilities in such dataset.

MAE is less sensitive to data with outliers

The lower value of MAE, MSE, and RMSE implies higher accuracy of a regression model. However, a higher value of R square is considered desirable.

R Squared and Adjusted R Squared are used for explaining how well the independent variables in the linear regression model explains the variability in the dependent variable.

R Squared value always increases with the addition of the independent variables which might lead to the addition of the redundant variables in our model. However, the adjusted R-squared solves this problem.

Adjusted R squared takes into account the number of predictor variables, and it is used to determine the number of independent variables in our model. The value of Adjusted R squared decreases if the increase in the R square by the additional variable isn't significant enough.

As a conclusion, for comparing the prediction accuracy among different linear regression models then RMSE/MSE is a better option than R-squared as it is simple to calculate and

differentiable. However, if your dataset has outliers then choose MAE over RMSE.

Example

Problem: Predicting House Prices

Scenario: Imagine you are a real estate agent and you want to predict house prices based on their size (in square feet). You have a dataset of houses with their sizes and corresponding prices.

Dataset:

Size (Square Feet)	Price (Thousands of Dollars)
1400	250
1600	300
1700	320
1875	370
1100	200

- **Objective:** Develop a linear regression model to predict house prices based on their size.
- **Linear Regression Model:** The linear regression model makes predictions by assuming a linear relationship between the input feature (size) and the output (price). The model can be represented by the equation:

$$\text{Price} = \theta_0 + \theta_1 \times \text{Size}$$

Here,

- θ_0 is the y-intercept (the price when the size is 0),
- θ_1 is the slope of the line (how much the price increases for each additional square foot).

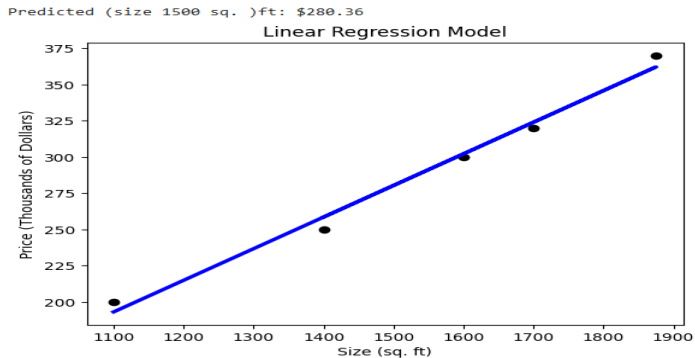
Training the Model: The goal is to find the values of θ_0 and θ_1 that minimize the difference between the predicted prices and the actual prices in the dataset. This is often done by minimizing the mean squared error. In this simple example, let's assume we already have the optimal parameters: $\theta_0=50, \theta_1=0.1$

Prediction: Now, given a new house with a size of 1500 square feet, we can use the model to predict its price:

$$\text{Price} = 50 + 0.1 \times 1500$$

$$\text{Price} = 50 + 150 = 200$$

So, the model predicts a price of \$200,000 for a house with a size of 1500 square feet. This is a basic example of linear regression, and in practice, the model would be trained using more sophisticated optimization algorithms to find the optimal parameters.



Sample Code

```

# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Input data
sizes = np.array([1400, 1600, 1700, 1875, 1100]).reshape(-1, 1)
# Reshape to a column vector
prices = np.array([250, 300, 320, 370, 200])

# Create and fit the linear regression model
model = LinearRegression()
model.fit(sizes, prices)

# Make predictions
new_size = np.array([1500]).reshape(-1, 1)
predicted_price = model.predict(new_size)

# Print the predicted price
print(f"Predicted (size 1500 sq. )ft: ${predicted_price[0]:.2f}")

# Plot the regression line
plt.scatter(sizes, prices, color='black')
plt.plot(sizes, model.predict(sizes), color='blue', linewidth=3)
plt.xlabel('Size (sq. ft)')
plt.ylabel('Price (Thousands of Dollars)')
plt.title('Linear Regression Model')
plt.show()

```

01

Parameters

The `LinearRegression()` model in scikit-learn has several parameters that you can use to customize its behavior. Here are some of the commonly used parameters:

1. **`fit_intercept (default=True):`**

- If True, the model includes an intercept term (0000) in the regression equation. If False, no intercept is used, and the regression line passes through the origin.
- 2. **normalize (default=False):**
 - If True and `fit_intercept` is True, the regressors (features) in `x` will be normalized before regression. This can be important when the features have different scales.
- 3. **copy_X (default=True):**
 - If True, the input data `x` is copied before fitting the model. If False, the model may modify `x` in place.
- 4. **n_jobs (default=None):**
 - The number of CPU cores to use during model fitting. If set to -1, it will use all available cores.
- 5. **positive (default=False):**
 - When set to True, forces the coefficients to be positive during fitting. Useful in situations where it makes sense for the relationship to be strictly positive.
- 6. **intercept_scaling (default=1.0):**
 - Affects the regularization term. It is only effective when the `fit_intercept` parameter is set to True.

These parameters can be set when creating an instance of `LinearRegression`. For example:

```
from sklearn.linear_model import LinearRegression

model = LinearRegression(fit_intercept=True, normalize=False, copy_X=True, n_jobs=None, positive=False, intercept_scaling=1.0)
```

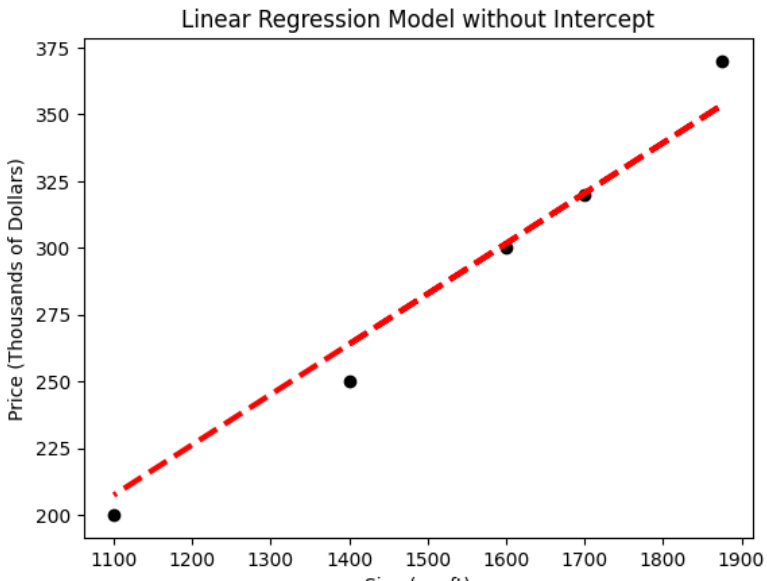
In practice, the most commonly used parameters are `fit_intercept` and `normalize`, depending on the characteristics of the

data. The default values often work well for many scenarios, and you may not need to modify them in most cases.

```
# Create and fit the linear regression model without intercept
model_without_intercept = LinearRegression(fit_intercept=False)
model_without_intercept.fit(sizes, prices)
```

In this example, we set `fit_intercept=False` when creating the `LinearRegression` model instance. As a result, the regression line will pass through the origin (0, 0), and the intercept term (0000) will be zero in the regression equation. This demonstrates how to use the `fit_intercept` parameter to control whether an intercept term is included in the linear regression model.

Predicted Price without intercept for a house with size 1500 sq. ft: \$282.77



Multiple Regression

In contrast to simple linear regression, multiple regression model extends to two or more independent explanatory variables are used to predict the value of a dependent variable (label), which can be expressed as follows:

$$y = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_ix_i = w^T x$$

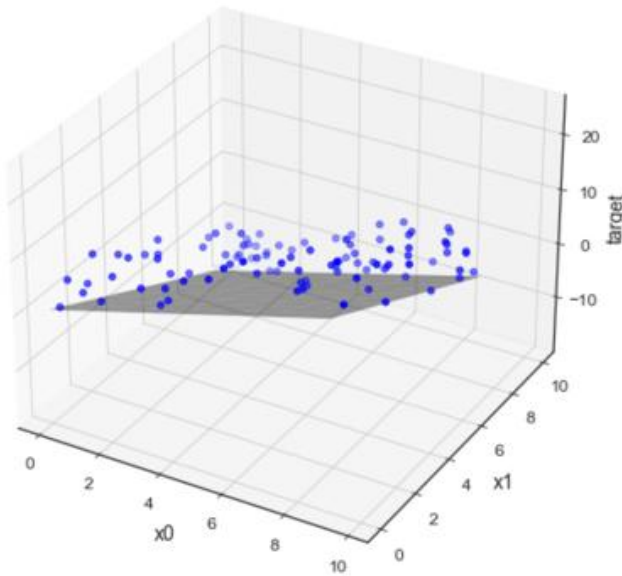
where

y = dependent variable

x_i = independent explanatory variables

w_0 = y-axis intercept (constant term) with $x_0 = 1$

w_1 = weight coefficients of the explanatory variable



The multiple regression model is based on the following assumptions:

- There is a **linear relationship between the dependent variables** and the **independent variables**
- The independent variables are not too highly correlated with each other

The model calculates the line of best fit that minimises the variances of each of the independent variables included as it relates to the dependent variable.

It is a linear model because it fits a line. There are also non-linear regression models involving multiple variables, such as polynomial regression, quadratic regression (polynomial of degree 2) and logistic regression models.

we extend the idea of linear regression to handle **multiple input features**. Let's consider an example where we want to predict the price of a house based on both its size (in square feet) and the number of bedrooms.

Sample code

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from mpl_toolkits.mplot3d import Axes3D

# Generate some example data
np.random.seed(42)

# Assume 100 houses with random sizes and number of bedrooms
sizes = np.random.randint(1000, 3000, 100)
bedrooms = np.random.randint(1, 5, 100)
prices = 50 * sizes + 30 * bedrooms + np.random.normal(0, 500, 100)
# Price = 50 * size + 30 * bedrooms + noise
```

```

# Create a DataFrame to organize the data
data = pd.DataFrame({'Size': sizes, 'Bedrooms': bedrooms, 'Price': prices})

# Split the data into training and testing sets
train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)

# Prepare the input features (X) and target variable (y) for training
X_train = train_data[['Size', 'Bedrooms']]
y_train = train_data['Price']

# Create and train the multiple linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions on the test set
X_test = test_data[['Size', 'Bedrooms']]
y_pred = model.predict(X_test)

```



```

# Evaluate the model performance
mse = mean_squared_error(test_data['Price'], y_pred)
print(f'Mean Squared Error on Test Set: {mse:.2f}')

# Visualize the data and regression plane
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot of the data points
ax.scatter(train_data['Size'], train_data['Bedrooms'], train_data['Price'],
           color='blue', marker='o', label='Training Data')
ax.scatter(test_data['Size'], test_data['Bedrooms'], test_data['Price'],
           color='red', marker='o', label='Testing Data')

# Create a meshgrid for the regression plane
size_range = np.linspace(min(data['Size']), max(data['Size']), 10)
bedroom_range = np.linspace(min(data['Bedrooms']), max(data['Bedrooms']), 10)
size_mesh, bedroom_mesh = np.meshgrid(size_range, bedroom_range)

```



```

# Predict prices for the meshgrid points
input_mesh = np.c_[size_mesh.ravel(), bedroom_mesh.ravel()]
price_mesh = model.predict(input_mesh)
price_mesh = price_mesh.reshape(size_mesh.shape)

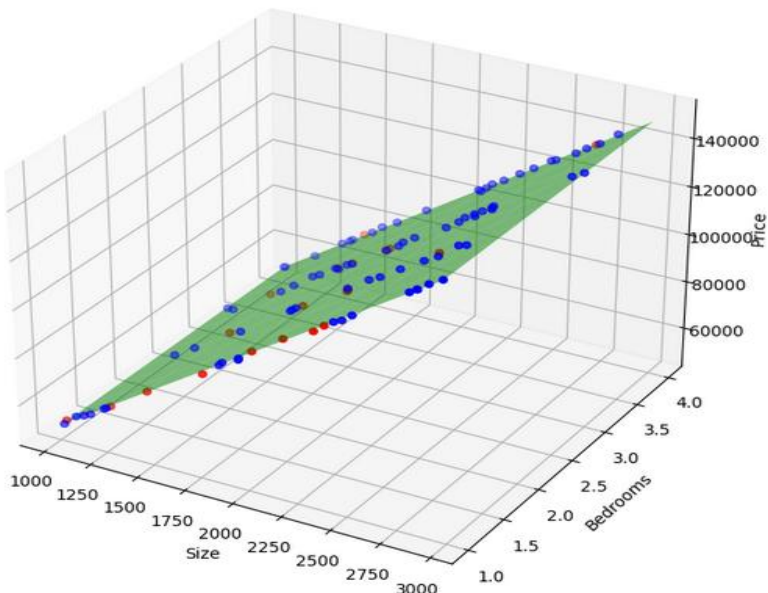
# Plot the regression plane
ax.plot_surface(size_mesh, bedroom_mesh, price_mesh, alpha=0.5,
               color='green', label='Regression Plane')

# Set axis labels
ax.set_xlabel('Size')
ax.set_ylabel('Bedrooms')
ax.set_zlabel('Price')
# Set plot title
plt.title('Multiple Linear Regression with 3D Visualization')

# Show the plot
plt.show()

```

Multiple Linear Regression with 3D Visualization



Polynomial Regression

Simple linear regression algorithm only works when the relation between dependent and independent variables is linear. But suppose if we have non-linear data then Linear regression will not be capable to draw a best-fit line in such conditions.

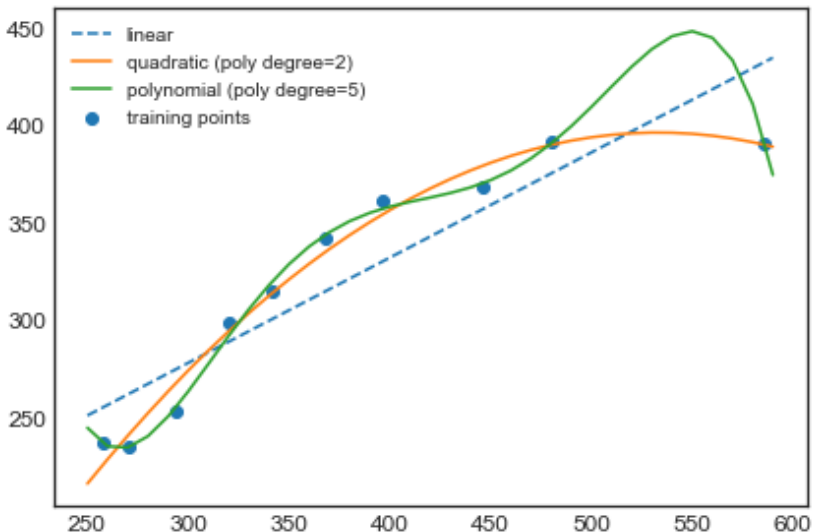
The problem can be overcome by adding polynomial terms to linear regression to convert it into Polynomial regression, which helps identify the curvilinear relation between independent and dependent variables.

In general, a polynomial regression of degree n has the formula of:

$$y = w_0 + w_1x + w_2x^2 \dots + w_dx^d$$

where d denotes the degree of the polynomial

Although, we can use polynomial regression to model a non-linear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients, w .



Example:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# Generate some example data
np.random.seed(42)

# Assume 50 data points with a quadratic relationship
X = np.sort(5 * np.random.rand(50, 1), axis=0)
y = 3 * X**2 + np.random.normal(0, 1, (50, 1))
# Quadratic relationship with noise

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)

# Polynomial features transformation (degree=2 for quadratic)
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly_features.fit_transform(X_train)
```

```

# Create and train the polynomial regression model
poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)

# Make predictions on the test set
X_test_poly = poly_features.transform(X_test)
y_pred = poly_model.predict(X_test_poly)

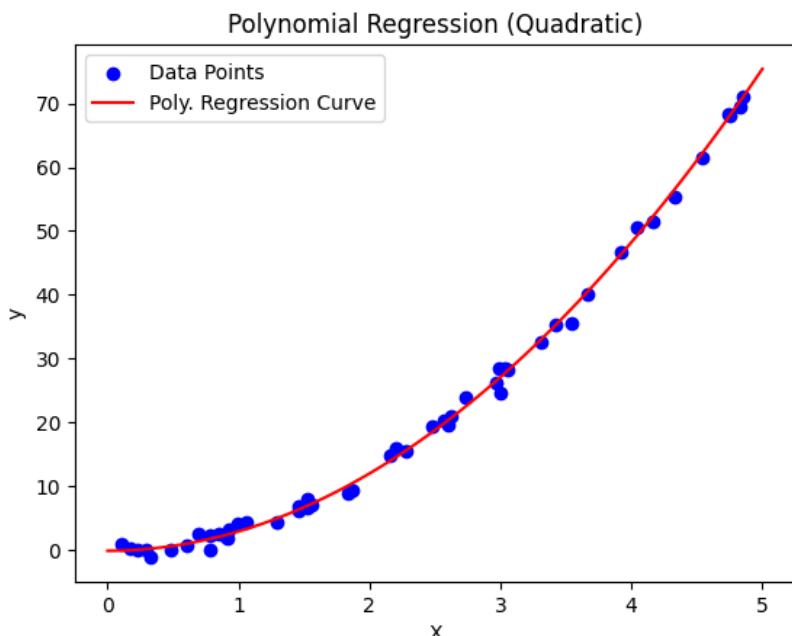
# Evaluate the model performance
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error on Test Set: {mse:.2f}')

# Visualize the data and polynomial regression curve
X_range = np.linspace(0, 5, 100).reshape(-1, 1)
X_range_poly = poly_features.transform(X_range)
y_range_pred = poly_model.predict(X_range_poly)

plt.scatter(X, y, color='blue', label='Data Points')
plt.plot(X_range, y_range_pred, color='red', label='Poly. Regression Curve')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Polynomial Regression (Quadratic)')
plt.legend()
plt.show()

```

Mean Squared Error on Test Set: 0.99



Example: Quadratic relationship

We generate synthetic data points with a quadratic relationship ($y=3X^2+\text{noise}$).

We use PolynomialFeatures from scikit-learn to transform the original feature (X) into polynomial features up to the 2nd degree.

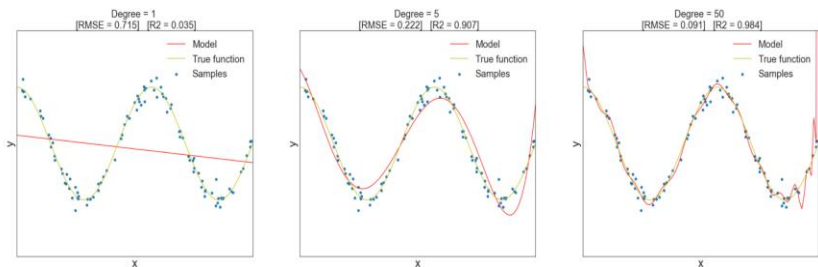
We train a linear regression model on the transformed features.

We make predictions on the test set and evaluate the model's performance using Mean Squared Error.

Finally, we visualize the original data points and the fitted quadratic polynomial regression curve.

You can modify the degree parameter in PolynomialFeatures to experiment with higher-degree polynomial regression. Keep in mind that increasing the degree might lead to overfitting, so it's essential to balance complexity and performance.

High-degree polynomial regression model has the tendency of falling into the common pitfalls **underfitting** and **overfitting**



The above model is considered simple because it is unable to capture non-linear representation. High bias and low variance
Producing high error on training set **underfitting**

Ideally, a ML model should have low bias and low variance. The above model is probably the best trade-off between bias and variance, hence performs well both on the train and unseen data – correct fitting

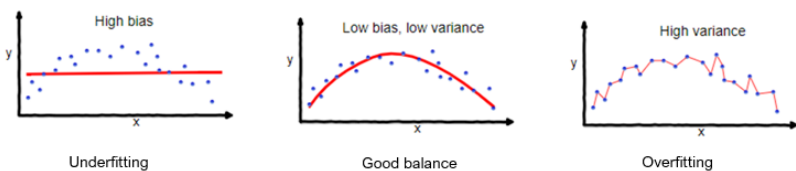
The above model is complex because it has tendency to accurately capture non-linear representation as well as capturing noises the dataset. Low bias and high variance High error on the test set – overfitting

Bias and Variance

Underfitting happens when a model unable to capture the underlying pattern of the data. It happens when we have very less amount of data to build an accurate model or when we try to build a linear model with a nonlinear data.

These models usually have high bias and low variance.

Overfitting happens when a model would not be able to generalise the data point in the test data set to predict the outcome accurately. it happens when we train our model a lot over noisy dataset. These models have low bias and high variance.

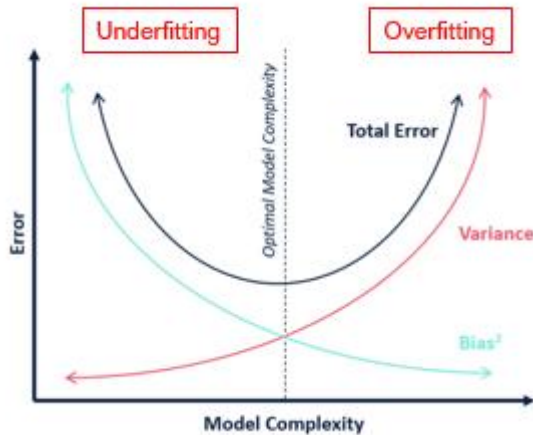


Bias and variance are some of those concepts that most of us forget to consider. What do they mean?

$$\text{Model Error} = \text{Bias} + \text{Variance} + \text{Inevitable Error}$$

The main reason for bias (or deviation) is that the assumption of the ML problem itself is incorrect, such as the use of linear regression for non-linear data. The model error is likely high bias and low variance that is generally known to be underfitting.

The main reason for the variance is that the model used is too complex, such as a high-order regression where a ML model is so precisely captures the position of each and every data point. The model error is likely low bias and high variance that is generally known to be overfitting.



Generally, a ML algorithm can be classified as either:

- Parametric ML algorithms that are inherently high-bias
- Non-parametric ML algorithms that are inherently high-variance

The trade-offs between parametric and non-parametric algorithms are in computational cost and accuracy

Parametric algorithm

- Uses a fixed number of parameters, in essence, this involves the following 2 steps:
 - Select a form for the function (linear or non-linear) as this determines the fixed number of parameters (aka. coefficients)
 - Learn the coefficients for the function from the training data

- Computationally faster
- The algorithm may work well if the assumptions turn out to be correct, but it may perform badly if the assumptions are wrong.
- Example parametric algorithms are Linear Regression, Logistic Regression, Naïve bayes, Neural Networks.

Non-parametric algorithm

- Uses a flexible number of parameters
- The number of parameters often grows as it learns from more data
- Computationally slower
- Example non-parametric algorithms are k-Nearest Neighbours (kNN), support vector machines (SVM), decision trees.

Regularized linear models

Linear regression works by selecting weights (coefficients) for each independent variable that minimises a cost function.

However, if the weights are too large, it can lead to model overfitting on the training dataset. Such a model will not generalize well on the unseen data. To overcome this shortcoming, we do regularization which penalizes large coefficients.

A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

Generally, for a linear model, regularization is typically achieved by constraining the weights of the model. There are 3 different ways to constrain the weights:

- Ridge Regression
- Lasso Regression
- Elastic Net

Hyper parameter Tuning and Model Performance

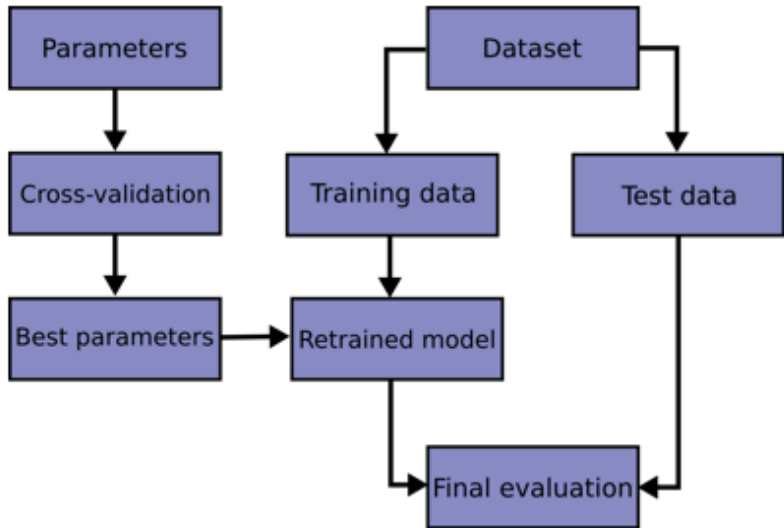
Learning the parameters of a prediction function and testing it on the same data is a methodological mistake

A model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting.

To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set X_{test} , y_{test} .

Note that the word “experiment” is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally.

Here is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by grid search techniques.



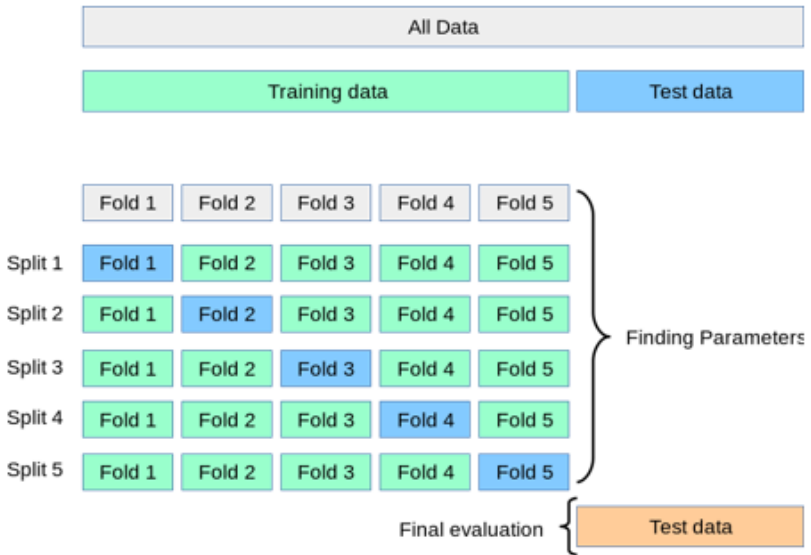
K-fold cross-validation

In k-fold cross-validation, the original sample is randomly partitioned into k equal sized subsamples.

Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k - 1$ subsamples are used as training data.

The cross-validation process is then repeated k times, with each of the k subsamples used exactly once as the validation data. The k results can then be averaged to produce a single estimation.

When the experiment seems to be successful, final evaluation can be done on Test data.



Grid Search Cross-Validation

The performance of a ML model significantly depends on the value of hyperparameters.

There is no way to know in advance the best values for hyperparameters. We need to try all possible values to know the optimal values.

Doing this manually could take a considerable amount of time and resources and thus we use GridSearchCV to automate the tuning of hyperparameters.

GridSearchCV is a function that comes in Scikit-learn's `model_selection` package. This function carrying out exhaustive search over a set of predefined hyperparameters and fit your estimator (model) on your training set, so in the end, we can select the best parameters from the listed hyperparameters.

Generally speaking, GridSearchCV is the process of performing hyperparameter tuning in order to determine the optimal values for a given model.

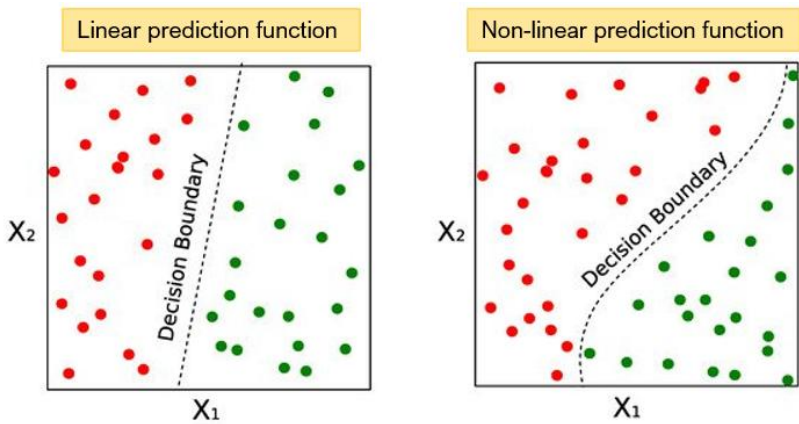
Classification

Classification is the process of predicting the class of given data points. Classes are sometimes called as targets/ labels or categories. Classification predictive modelling is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y). There are many different types of classification tasks that you may encounter in machine learning and specialised approaches to modelling that may be used for each. There are perhaps four main types of classification tasks that you may encounter; they are:

- Binary Classification
- Multi-Class Classification
- Multi-Label Classification
- Imbalanced Classification

Binary Classification

Binary classification refers to those classification tasks that have two class labels. Examples include: Email spam detection (spam or not), Churn prediction (churn or not), Conversion prediction (buy or not) Typically, binary classification tasks involve one class that is the normal state and another class that is the abnormal state. The class for the normal state is assigned the class label 0 (Red) and the class with the abnormal state is assigned the class label 1 (Green).



It is common to model a binary classification task with a model that predicts a Bernoulli probability distribution for each example. The Bernoulli distribution is a discrete probability distribution that covers a case where an event will have a binary outcome as either a 0 or 1. For classification, this means that the model predicts a probability of an example belonging to class 1, or the abnormal state.

Popular algorithms that can be used for binary classification include:

- Logistic Regression
- k-Nearest Neighbours
- Decision Trees
- Support Vector Machine (SVM)
- Random Forest

Multiclass Classification

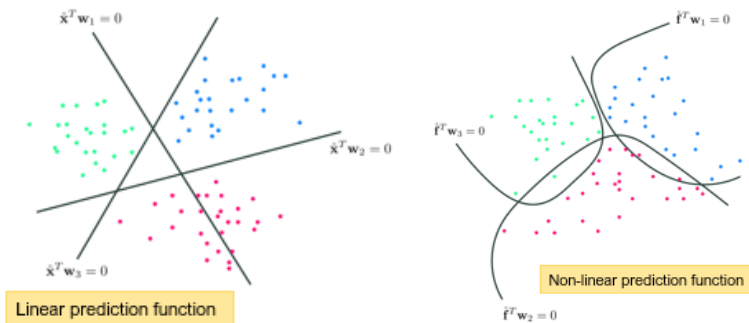
Multi-class classification refers to those classification tasks that have more than two class labels. Examples include:

Face classification, Plant species classification, Optical character recognition

Unlike binary classification, multi-class classification does not have the notion of normal and abnormal outcomes. Instead, examples are classified as belonging to one among a range of known classes.

The number of class labels may be very large on some problems.

For example, a model may predict a photo as belonging to one among thousands or tens of thousands of faces in a face recognition system.



It is common to model a multi-class classification task with a model that predicts a Multinomial probability distribution for each example.

The Multinomial distribution is a discrete probability distribution that covers a case where an event will have a categorical outcome, e.g. K in $\{1, 2, 3, \dots, K\}$. For classification, this means that the model predicts the probability of an example belonging to each class label.

Many algorithms used for binary classification can be used for multi-class classification, for examples:

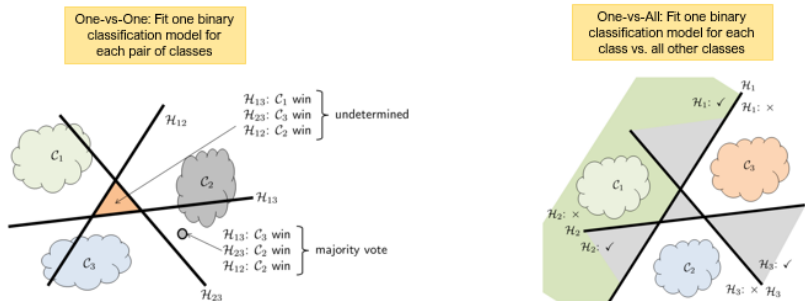
- Logistic Regression
- k-Nearest Neighbours
- Decision Trees
- Support Vector Machine (SVM)
- Random Forest

Algorithms that are designed for binary classification can be adapted for use for multi-class problems.

This involves using a strategy of fitting multiple binary classification models, namely:

One-vs-One: Fit one binary classification model for each pair of classes

One-vs-All: Fit one binary classification model for each class vs. all other classes



One-vs-One strategy introduced $(k-1)/2$ binary classifiers, one for each possible pair of classes. For an example, consider a 3-classes classification problem: 'red,' 'blue,' and 'green', the following comparisons are required:

Binary Classification #1: Red vs. Blue

Binary Classification #2: Red vs. Green

Binary Classification #3: Blue vs. Green

Each point is classified according to a majority vote amongst the discriminant function as follows:

$$\mathcal{H}_{ij}(x) = \mathcal{H}_i(x) - \mathcal{H}_j(x) \leq 0 \quad \text{majority wins}$$

This strategy is not suitable for classification problem with large number of classes and large training dataset. Let say 100 classes, it will need 4950 classifiers that is a huge computation resource.

This strategy is suggested for the implementation of Support Vector Machines (SVM) and related kernel-based algorithms, which normally used small classes and dataset.

For a k -class problem, One-vs-All only requires number of binary classifiers (or more for better precision). For example, consider a 3-classes classification problem ‘red,’ ‘blue,’ and ‘green’, the following comparisons are required :

- Binary Classification #1: red vs [blue, green]
- Binary Classification #2: blue vs [red, green]
- Binary Classification #3: green vs [red, blue]

This strategy is far more effective and scalable; for 100-classes problem, one-vs-all only requires 100 classifiers as compared to 4950 classified required by the one-vs-one.

Each classifier partitions the hyperplane space with a decision boundary by checking data point $x \in C_i$ versus $x \notin C_i$, or the following hyperplane check rule :

It may have multiple undetermined regions e.g. the region “A” because is not allowed by both C_1 and C_3 . Similarly, the centre triangle is also ambiguous as it is not allowed by any of the classes.

This strategy is commonly used for algorithms that naturally predict numerical class membership probability such as Perceptron or Logistic Regression.

Multi-label Classification

Multi-label classification refers to those classification tasks that have two or more class labels, where one or more class labels may be predicted for each example.

Consider the example of photo classification, where a given photo may have multiple objects in the scene and a model may predict the presence of multiple known objects in the photo, such as “cat”, “bird”, “bicycle,” “apple,” “person,” etc.

This is unlike binary classification and multi-class classification, where a single class label is predicted for each example.

Algorithms used for binary or multi-class classification cannot be used directly for multi-label classification. Specialised versions of standard classification algorithms can be used, so-called multi-label versions of the algorithms, including:

- Multi-label Decision Trees
- Multi-label Random Forests
- Multi-label Gradient Boosting



Imbalance Classification

Imbalanced classification refers to classification tasks where the number of examples in each class is unequally distributed.

Typically, imbalanced classification tasks are binary classification tasks where the majority of examples in the training dataset belong to the normal class and a minority of examples belong to the abnormal class.

Specialised techniques may be used to change the composition of samples in the training dataset by undersampling the majority class or oversampling the minority class. Examples include:

Random Undersampling.

SMOTE Oversampling.

Specialised modelling algorithms may be used that pay more attention to the minority class when fitting the model on the training dataset, such as cost-sensitive machine learning algorithms. Examples include:

- Cost-sensitive Logistic Regression
- Cost-sensitive Decision Trees

- Cost-sensitive Support Vector Machines

Examples include:

Fraud detection, Outlier detection, Medical diagnostic tests,

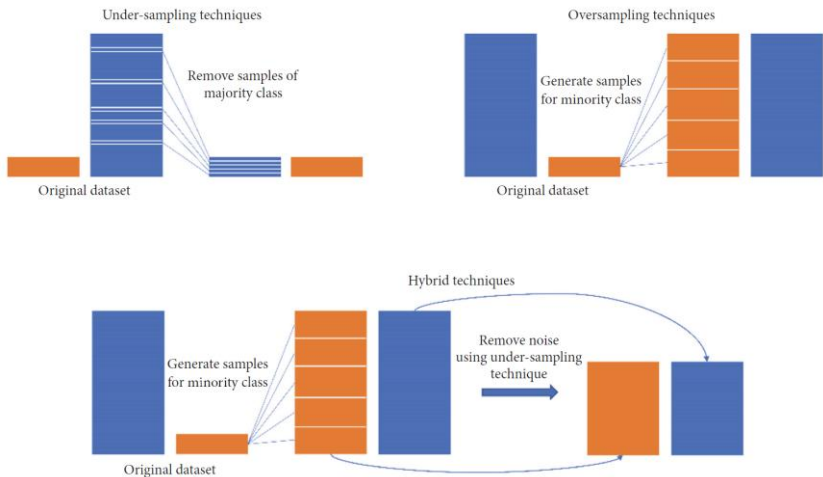


FIGURE 1: The illustration of oversampling, undersampling, and hybrids techniques.

Classification Performance Measures

There are many performance metrics/measures:

- Confusion Matrix
- 4 key classification metrics
 - Accuracy
 - Precision
 - Recall
 - F1-Score
- Area Under Receiver Operating Characteristics Curve (AUROC)

Confusion Matrix

True positives (TP)

The cases when the actual class of the data point was 1(True) and the predicted is also 1(True).

		Actual class		
		Positive	Negative	
Predicted class	Positive	TP: True Positive	FP: False Positive (Type I Error)	Precision: $\frac{TP}{TP + FP}$
	Negative	FN: False Negative (Type II Error)	TN: True Negative	Negative Predictive Value: $\frac{TN}{TN + FN}$
		Recall or Sensitivity: $\frac{TP}{TP + FN}$	Specificity: $\frac{TN}{TN + FP}$	Accuracy: $\frac{TP + TN}{TP + TN + FP + FN}$

False Positive (FP)

The cases when the actual class of the data point was 0(False) and the predicted is 1(True). False is because the model has predicted incorrectly and positive because the class predicted was a positive one (1).

False negatives (FN)

The cases when the actual class of the data point was 1(True) and the predicted is 0(False). False is because the model has predicted incorrectly and negative because the class predicted was a negative one (0).

True Negative (TN)

The cases when the actual class of the data point was 0(False) and the predicted is also 0(False).

Confusion Matrix: Accuracy

Accuracy in classification problems is the number of correct predictions made by the model over all kinds predictions made.

Use case:

Out of all the patients who visited the doctor, how many were correctly diagnosed as Covid positive and Covid negative.

When to use:

Accuracy is a good measure when the target variable classes in the data are nearly balanced.

		Actual class		
		Positive	Negative	
Predicted class	Positive	TP: True Positive	FP: False Positive (Type I Error)	Precision: $\frac{TP}{TP + FP}$
	Negative	FN: False Negative (Type II Error)	TN: True Negative	Negative Predictive Value: $\frac{TN}{TN + FN}$
		Recall or Sensitivity: $\frac{TP}{TP + FN}$	Specificity: $\frac{TN}{TN + FP}$	Accuracy: $\frac{TP + TN}{TP + TN + FP + FN}$

Accuracy in classification problems is the number of correct predictions made by the model over all kinds predictions made.

Accuracy metric is not very useful in the case of highly imbalanced training dataset (e.g. 95% positive class and 5% negative class)

This is because a ML model will end up learning how to predict the positive class properly and will not learn how to identify the negative class. But the model will still have a very high accuracy in the test dataset too as it will know how to identify the positives really well.

Confusion Matrix: Precision

Precision is the ratio of TP to all the predicted positives (TP + FP) by the model.

In other words, precision tells us how many cases are actually truly positive out of all that were predicted positive.

Precision metric is used when we want our ML model to minimise False Positive (FP), and in this case the Precision should be as close to 100% as possible.

		Actual class		
		Positive	Negative	
Predicted class	Positive	TP: True Positive	FP: False Positive (Type I Error)	Precision: $\frac{TP}{TP + FP}$
	Negative	FN: False Negative (Type II Error)	TN: True Negative	Negative Predictive Value: $\frac{TN}{TN + FN}$
		Recall or Sensitivity: $\frac{TP}{TP + FN}$	Specificity: $\frac{TN}{TN + FP}$	Accuracy: $\frac{TP + TN}{TP + TN + FP + FN}$

Confusion Matrix: Recall

Recall is the ratio of TP to all the real positive cases (TP + FN) by the model. In other words, Recall tells us how many are actually truly positive out of all that were actually positive including both the predicted positive and those falsely predicted that were actually positive. Recall metric is used when we want our ML model to minimise False Negative (FN), and in this case the Recall should be as close to 100% as possible.

		Actual class		
		Positive	Negative	
Predicted class	Positive	TP: True Positive	FP: False Positive (Type I Error)	Precision: $\frac{TP}{(TP + FP)}$
	Negative	FN: False Negative (Type II Error)	TN: True Negative	Negative Predictive Value: $\frac{TN}{(TN + FN)}$
		Recall or Sensitivity: $\frac{TP}{(TP + FN)}$	Specificity: $\frac{TN}{(TN + FP)}$	Accuracy: $\frac{TP + TN}{(TP + TN + FP + FN)}$

Confusion Matrix: Specificity

Specificity is the opposite of Recall that the ratio of TN to all the real negative cases (TN + FP) by the model.

In other words, Specificity tells us how many are actually truly negative out of all that were actually negative including both the predicted negative and those falsely predicted that were actually negative.

		Actual class		
		Positive	Negative	
Predicted class	Positive	TP: True Positive	FP: False Positive (Type I Error)	Precision: $\frac{TP}{(TP + FP)}$
	Negative	FN: False Negative (Type II Error)	TN: True Negative	Negative Predictive Value: $\frac{TN}{(TN + FN)}$
Recall or Sensitivity:		$\frac{TP}{(TP + FN)}$	Specificity: $\frac{TN}{(TN + FP)}$	Accuracy: $\frac{TP + TN}{(TP + TN + FP + FN)}$

F1 Score

F1 score is one of the common measures to rate how successful a classifier is. It is very useful metric compared to “Accuracy”.

F1 score is the harmonic mean of precision and recall, which means there is equal importance given to FP and FN.

In a binary classification problem, the formula is:

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

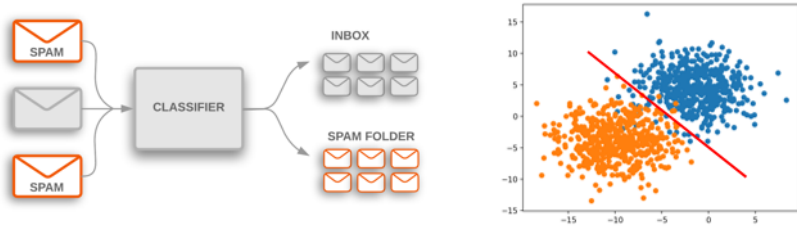
For a multi-class classification problem, we don’t calculate an overall F1 score. Instead, we calculate the F1 score per class in a one-vs-all manner.

$$\begin{aligned} &\text{F1 Score}(\text{class} = x) \\ &= \frac{2 \times \text{Precision}(\text{class} = x) \times \text{Recall}(\text{class} = x)}{\text{Precision}(\text{class} = x) + \text{Recall}(\text{class} = x)} \end{aligned}$$

Logistic Regression

Logistic regression is known and used as a linear classifier, but instead of the continuous output, it is regressing for the probability of a categorical outcome.

For example, what is the probability that this email is spam?



If the estimated probability is greater than 50% ($=0.5$), then the model predicts that the instance belongs to that class (called the positive class, labelled “1”), or else it predicts that it does not (i.e., it belongs to the negative class, labelled “0”). This makes it a binary classifier.

The basic idea of logistic regression is to use linear predictor function of linear regression to estimate the probability of a random variable being 0 or 1 for a given set of observation.

With such aspect in mind, the linear output of the logistic regression must be bounded between 0 and 1.

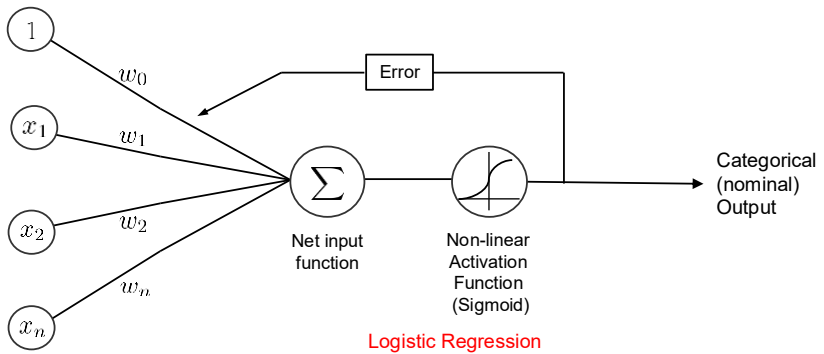
We can use Sigmoid function (a.k.a Logistic function), which takes any real value input, and outputs a value between 0 and 1. Sigmoid (logistic) function forces the output to assume only values between 0 and 1:

Logistic Regression: Model

As the logistic regression linear predictor function outputs a number between 0 and 1, we can create the following rules:

If $p = h_{\theta}(x) \geq 0.5$ predict $y = 1$

If $p = h_{\theta}(x) < 0.5$ predict $y = 0$



In this case, we can formally formulate a probability function for predicting y given x parameterised by θ as follow:

Since the prediction is binary case where y is either 0 or 1, we can apply Bernoulli Trials (1 trial) to model the probability function as follows, also called likelihood function:

Logistic Regression: Loss function

Here, we aim to estimate an unknown parameter θ by the value for which the likelihood function is maximum by applying natural log, this approach is called *maximum-likelihood estimation (MLE)* as below:

In ML optimisation, we prefer to use gradient descent instead of ascent to find the optimum point. We do this because the learning/optimising of neural networks is posed as a “minimisation of loss” problem, so this is where we add the negative sign to the log of Bernoulli distribution, the result is the Binary Cross-Entropy Loss function:

Cost Function

Finally, we divide the negative log-likelihood loss function by the total number of samples (m), and we obtain Cost function as follows:

Nonlinear Logistic Regression

We can easily model the logistic regression as a nonlinear classifier, by simply extending the linear equation (see slide 28) into higher degree polynomial equation:

Note that the above equation is identical to the linear case, if we treat x , x^2 , $x^3 \dots x^d$ as additional independent features of the problem.

Therefore, we can then directly apply the scikit-learn's LogisticRegression function to get an estimate for the parameters (w).

Having said that, we need to use scikit-learn's PolynomialFeatures function to create polynomial variables (x , x^2 , $x^3 \dots x^d$) as the additional independent features.

Example

Logistic Regression is a classification algorithm used to model the probability of a binary outcome. In this example, let's consider a simple case where we want to predict whether a student passes (1) or fails (0) an exam based on the number of hours they studied.

```

# Import necessary libraries
import numpy as np
import pandas as pd # Don't forget to import pandas
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix

# Generate some example data
np.random.seed(42)

# Assume 100 students with a binary outcome (pass/fail) based on hours studied
hours_studied = np.random.uniform(0, 10, 100)
pass_probability = 1 / (1 + np.exp(-0.8 * (hours_studied - 5)))
# Sigmoid function
pass_status = np.random.binomial(1, pass_probability)

# Create a DataFrame to organize the data
data = {'Hours_Studied': hours_studied, 'Pass_Status': pass_status}
df = pd.DataFrame(data)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df[['Hours_Studied']],
                                                    df['Pass_Status'], test_size=0.2, random_state=42)

# Create a DataFrame to organize the data
data = {'Hours_Studied': hours_studied, 'Pass_Status': pass_status}
df = pd.DataFrame(data)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df[['Hours_Studied']],
                                                    df['Pass_Status'], test_size=0.2, random_state=42)

# Create and train the logistic regression model
logistic_model = LogisticRegression()
logistic_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = logistic_model.predict(X_test)

# Evaluate the model performance
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f'Accuracy: {accuracy:.2f}')
print('Confusion Matrix:')
print(conf_matrix)

```

```

# Visualize the data and decision boundary
plt.scatter(df['Hours_Studied'], df['Pass_Status'],
            color='blue', marker='o', label='Data Points')
plt.xlabel('Hours Studied')
plt.ylabel('Pass Status')
plt.title('Logistic Regression Example')
plt.legend()

# Plot the decision boundary
x_range = np.linspace(0, 10, 100).reshape(-1, 1)
decision_boundary = logistic_model.predict_proba(x_range)[: , 1]
plt.plot(x_range, decision_boundary, color='red',
        label='Decision Boundary')
plt.legend()
plt.show()

```

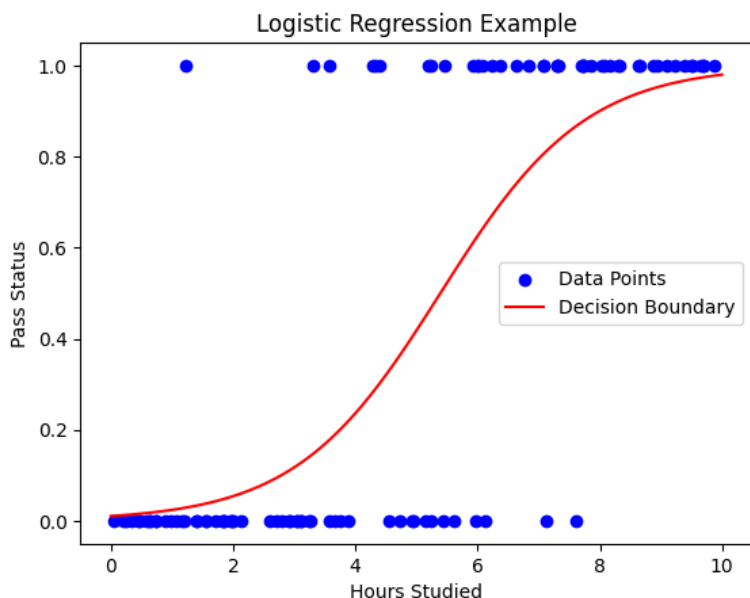
Accuracy: 0.90

Confusion Matrix:

```

[[9 0]
 [2 9]]

```



- We generate synthetic data for 100 students with a binary outcome (pass/fail) based on the number of hours they studied.
- We use the logistic function (sigmoid function) to model the probability of passing.
- We split the data into training and testing sets.
- We create and train a logistic regression model using scikit-learn.
- We make predictions on the test set and evaluate the model's accuracy and confusion matrix.
- Finally, we visualize the data points and the decision boundary of the logistic regression model.

Note: It's important to preprocess and clean real-world data before applying logistic regression. This example is kept simple for illustrative purposes.

Nonlinear logistic regression involves using features transformed by nonlinear functions, making the decision boundary more flexible. A common approach is to use polynomial features in logistic regression. In this example, let's consider a case where we have a nonlinear relationship between the hours studied and pass status, modeled by a quadratic polynomial.

Support Vector Machine

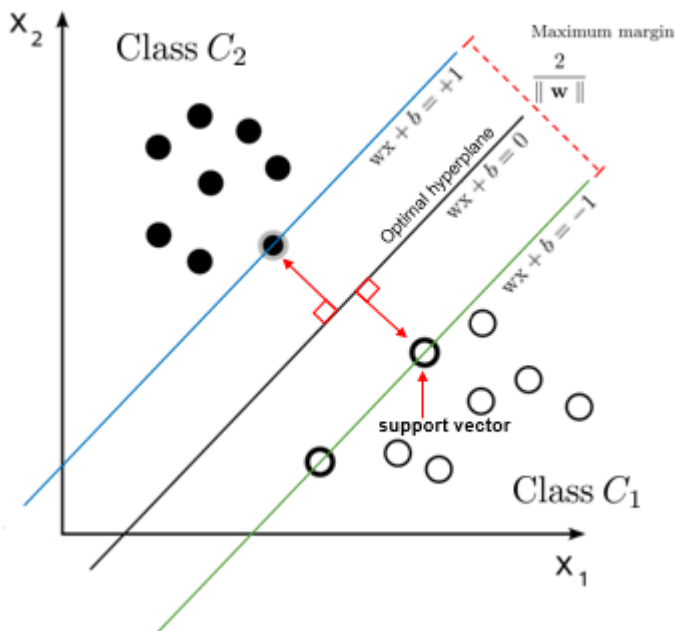
Support Vector Machine (SVM) is arguably one of the most popular ML algorithm for numerical prediction and classification tasks.

Terminology:

Support vectors are the data points, which are closest to the **hyperplane**. These points will define the separating line better by calculating margins. These points are more relevant to the construction of the classifier.

Hyperplane is a decision plane which separates between a set of objects having different class memberships.

Margin is a gap between the two lines on the closest class points. This is calculated as the perpendicular distance from the line to support vectors or closest points. If the margin is larger in between the classes, then it is considered a good margin, a smaller margin is a bad margin.



SVM operates by generating optimal hyperplane (decision boundary) in an iterative manner, with the objective to:

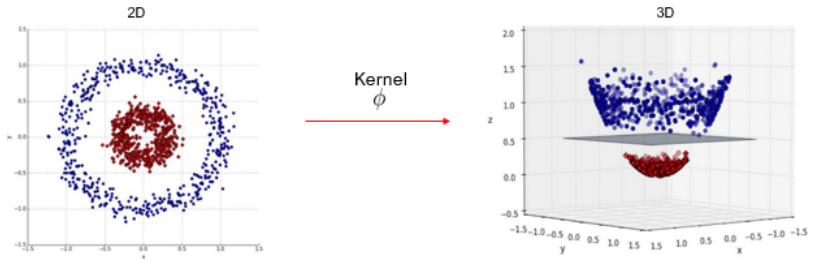
maximise the margin, the distance between two decision boundaries established by the support vectors belonging to opposite classes, that best divides the dataset into classes

minimise the prediction error between each optimal hyperplane it generated

If the data is not linearly separable in the original feature space, we can apply transformations to the data, which map the data from the original space into a higher dimensional feature space using one of many SVM's kernels namely linear, polynomial, radial basis function, and sigmoid.

After the transformation, the classes are now linearly separable in this higher dimensional feature space.

We can then fit a decision boundary (hyperplane) in this higher dimensional space to separate the classes and make predictions.



For data not easily separable in 2D feature space, we need to transform the data into higher dimension by applying one of many SVM's nonlinear kernels.

The nonlinear kernel takes input features and transform it into the required form to achieve higher dimensions – this process is often known as Kernel Trick

Example kernels are:

$$K_{\text{poly}}(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j)^d$$

$$K_{\text{rbf}}(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$$

$$K_{\text{sigmoid}}(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i^\top \mathbf{x}_j + r)$$

C is a hyperparameter to adjust the constraints of the support vector boundaries (+1,-1) that is used to control error.

- Low C means a wide margin is selected at the cost of a larger number of misclassifications.
- High C means a narrow margin is selected aims to reduce the number of misclassified.

Gamma is a hyperparameter used when using RBF or Sigmoid, to control how much curvature we want in a decision boundary.

- High Gamma means more curvature
- Low Gamma means less curvature

Margin classification, **C and Gamma**

- Margin Classification – by now it is apparent that SVM classification relies on the margin.
- Hard margin is very sensitive to outliers and can easily result in overfitting.
- Soft margin is a more flexible model, allow to make errors in decision, it produces a more generalisable model.
- The hard/soft margin can be controlled using the C and Gamma hyperparameters.

C is to adjust the constraints of the support vector boundaries (+1,-1) that is used to control error.

- Low C means a wide margin is selected at the cost of a larger number of misclassifications.
- High C means a narrow margin is selected aims to reduce the number of misclassified.

Gamma is used when using RBF or Sigmoid, to control how much curvature we want in a decision boundary.

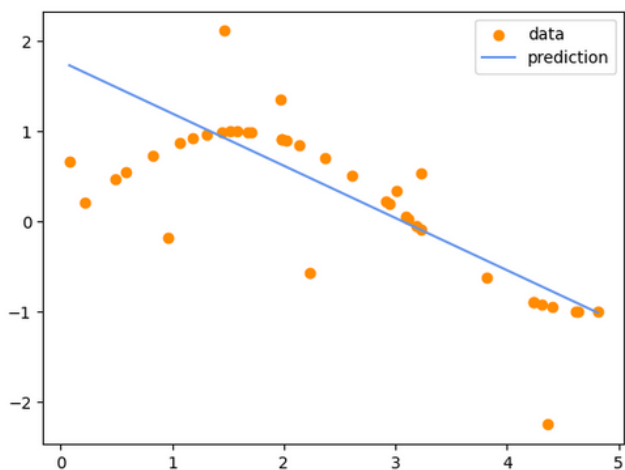
- High Gamma means more curvature
- Low Gamma means less curvature

Linear Kernel

Fitting an SVR Model on the Sine Curve data using Linear Kernel
First, we will try to achieve some baseline results using the linear kernel on a non-linear dataset and we will try to observe up to what extent it can be fitted by the model.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR

# generate synthetic data
X = np.sort(5 * np.random.rand(40, 1),
            axis=0)
y = np.sin(X).ravel()
# add some noise to the data
y[::5] += 3 * (0.5 - np.random.rand(8))
# create an SVR model with a linear kernel
svr = SVR(kernel='linear')
|
# train the model on the data
svr.fit(X, y)
# make predictions on the data
y_pred = svr.predict(X)
# plot the predicted values against the true values
plt.scatter(X, y, color='darkorange',
            label='data')
plt.plot(X, y_pred, color='cornflowerblue',
         label='prediction')
plt.legend()
plt.show()
```



Polynomial Kernel

Fitting an SVR Model on the Sine Curve data using Polynomial Kernel Now we will fit a Support vector Regression model using a polynomial kernel. This will be hopefully a little better than the SVR model with a linear kernel.

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR

# generate synthetic data
X = np.sort(5 * np.random.rand(40, 1), axis=0)
y = np.sin(X).ravel()

# add some noise to the data
y[::5] += 3 * (0.5 - np.random.rand(8))

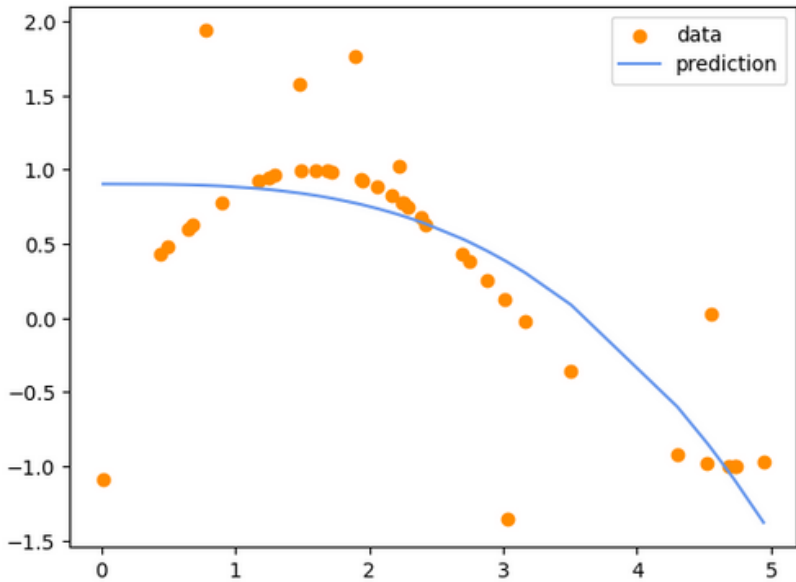
# create an SVR model with a linear kernel
svr = SVR(kernel='poly')

# train the model on the data
svr.fit(X, y)

# make predictions on the data
y_pred = svr.predict(X)

# plot the predicted values against the true values
plt.scatter(X, y, color='darkorange',
            label='data')
plt.plot(X, y_pred, color='cornflowerblue',
         label='prediction')
plt.legend()
plt.show()

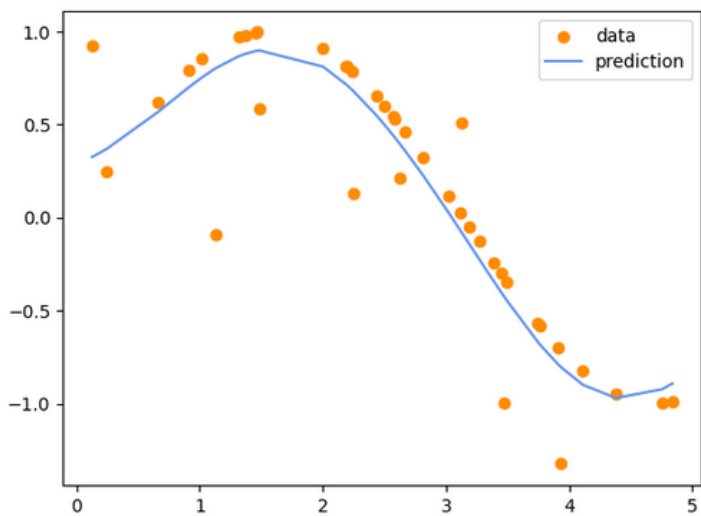
```



RBF Kernel

Fitting an SVR Model on the Sine Curve data using RBF Kernel
Now we will fit a Support vector Regression model using an RBF(Radial Basis Function) kernel. This will help us to achieve probably the best results as the RBF kernel is one of the best kernels which helps us to introduce non-linearity in our model.

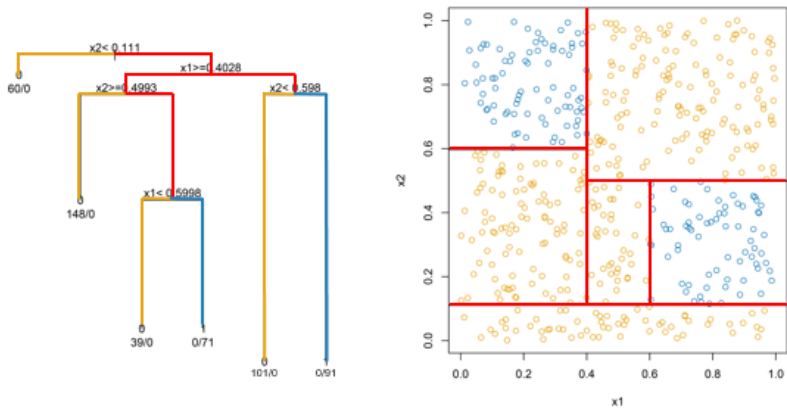
```
# create an SVR model with a linear kernel  
svr = SVR(kernel='rbf')
```



Decision Tree

Decision tree algorithms operate by dividing feature space into subspaces (i.e. rectangles) in iteration, top-down construction of the hypothesis, and a hierarchy of decision into a tree.

In other words, a tree is built up via a greedy algorithm i.e. recursive binary partitioning



Terminology:

Root node: No incoming edge, zero or more outgoing edges.

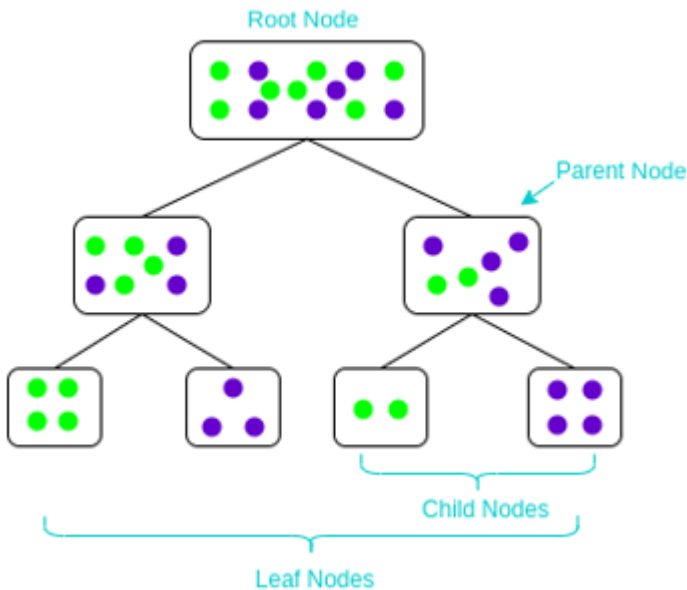
Parent and child nodes: A node that gets divided into sub-nodes are called Parent Node, and its sub-nodes are called Child Nodes, respectively.

Leaf node: Nodes that do not have any child node are called Terminal/Leaf Nodes. Each leaf node is assigned a class label (if nodes are pure; otherwise majority vote).

Node splitting is the process of dividing a node into multiple sub-nodes to create relatively pure nodes, based on two categories of splitting criterion depending on the target variable type:

- Categorical
 - Information Entropy and Information Gain
 - Gini Impurity
- Continuous
 - Reduction in Variance

Once the tree is trained, a new instance is classified by starting at the root and following the path as dictated by the test results for this instance.



Explain ability

Decision Tree is categorised as a supervised ML (classification and regression), optimisation method (combinatorial -> greedy search), Nonparametric model, Deterministic, Eager learning (vs Lazy learning), and Batch learning.

It is simple and easily explainable (interpretable) how the classification decisions were made.

The downside is that with the greater explainability comes a loss in accuracy when compared to these other techniques (see below).

Tree growing algorithm

- The process of growing a decision tree can be expressed as a recursive algorithm as follows:
- Pick a feature such that when parent node is split, it results in the largest information gain.
- Stop if child nodes are pure or no improvement in class purity can be made (information gain ≤ 0)
- Go back to step 1 for each of the two child nodes.
- Below is a more formal expression of the algorithm:

Some design choices such as:

- How to split?
- What splitting criterion to measure the goodness of the split?
- Binary or multi-classes split
- When to stop?
- If leaf nodes contain only examples of the same class
- Feature values are all the same for all examples
- Statistical significance test
- How do we make predictions of features in dataset not sufficient to make child nodes pure?

Three well-known tree growing algorithms:

- ID3 (Iterative Dichotomiser 3)
- CAT 4.5 (computerized adaptive testing 4.5)
- CART (Classification and Regression Trees)

ID3 (Iterative Dichotomiser 3)

- One of the earlier/earliest decision tree algorithms By Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1 (1), 81-106.
- Discrete features: binary and multi-category features
- Cannot handle numeric features
- No pruning, prone to overfitting
- Short and wide trees (compared to CART)
- Splitting criterion: Shannon's information entropy and information gain

CAT 4.5 (computerized adaptive testing 4.5)

- Discrete and Continuous features
- By Quinlan, J. R. (1993). C4.5: Programming for machine learning. Morgan Kaufmann, 38, 48.
- Continuous feature splitting is very expensive because must consider all possible ranges
- Handles missing attributes (ignores them in information gain computation)
- Performs post-pruning (bottom-up pruning)
- Splitting criterion: gain ratio

CART (Classification and Regression Trees)

- By Breiman, L. (1984). Classification and regression trees. Belmont, Calif: Wadsworth International Group
- Discrete and Continuous features
- Strictly binary splits (resulting trees are taller compared to ID3 and C4.5)

- Binary splits can generate better trees than C4.5, but tend to be larger and harder to interpret; i.e., for k attributes, we have $2^k - 1$ ways to create a binary partitioning
- Performs cost-complexity pruning
- Splitting criterion:
- Gini impurity for classification trees
- Variance reduction for regression trees

C5.0 (patented)

A successor of C4.5 algorithm also developed by Quinlan (1994).

Uses Information Gain (Entropy) as its splitting criteria.

C5.0 pruning technique adopts the Binomial Confidence Limit method.

CHAID (CHi-squared Automatic Interaction Detector); Kass, G. V. (1980). “An exploratory technique for investigating large quantities of categorical data.” *Applied Statistics*. 29 (2): 119–127.

MARS (Multivariate adaptive regression splines); Friedman, J. H. (1991). “Multivariate Adaptive Regression Splines.” *The Annals of Statistics*. 19: 1

Most decision tree algorithms differ in the following ways:

- Splitting criterion: information gain (Shannon’s Information Entropy, Gini impurity, misclassification error), use of statistical tests, objective function, etc.
- Binary split vs. multi-way splits
- Discrete vs. continuous variables
- Pre- vs. post-pruning

Below are some of the pros and cons of using decision trees as a predictive model.

Pros:

- Easy to interpret and communicate
- Independent of feature scaling

Cons:

- Easy to overfit
- Elaborate pruning required
- Expensive to just fit a “diagonal line”
- Output range is bounded (dep. on training examples) in regression trees

Example

```

# Import necessary libraries
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a toy dataset for binary classification
X, y = make_classification(n_samples=100, n_features=2, n_classes=2,
                           n_informative=2, n_redundant=0, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Create and train the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

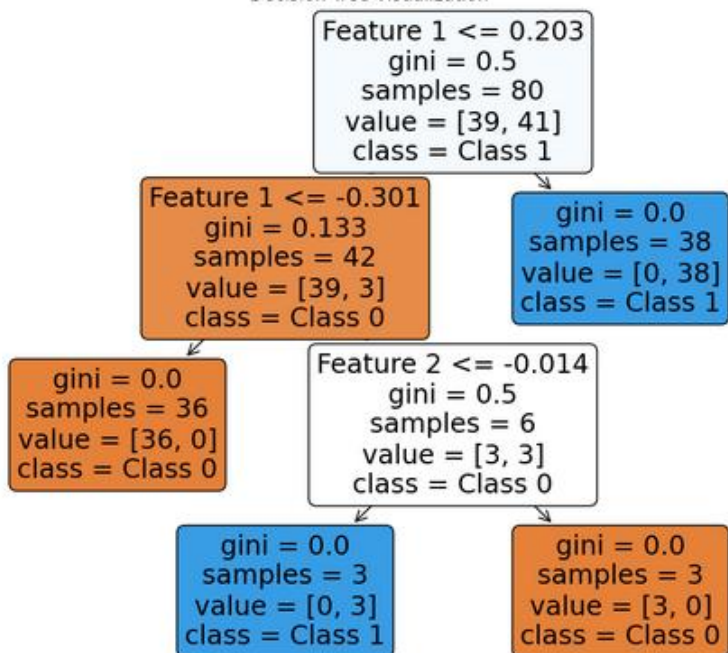
# Make predictions on the test set
y_pred = dt_model.predict(X_test)

# Evaluate the model performance
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Visualize the Decision Tree
plt.figure(figsize=(10, 8))
plot_tree(dt_model, filled=True, feature_names=['Feature 1', 'Feature 2'],
          class_names=['Class 0', 'Class 1'], rounded=True)
plt.title('Decision Tree Visualization')
plt.show()

```

Decision Tree Visualization



Ensemble Learning Methods

Ensemble methods can be classified into two groups based on how the sub-learners are generated:

Sequential ensemble method

Learners are generated sequentially.

These methods use the dependency between base learners. Each learner influences the next one, likewise, a general paternal behaviour can be deduced.

A popular example of sequential ensemble algorithms is AdaBoost (or more advanced variance XGBoost)

Parallel ensemble method

Learners are generated in parallel.

The base learners are created independently to study and exploit the effects related to their independence and reduce error by averaging the results.

An example implementing this approach is Random Forest.

Ensemble methods can use homogeneous learners (learners from the same family) or heterogeneous learners (learners from multiple sorts, as accurate and diverse as possible).

Homogeneous ensembles have a single-type base learning algorithm, in which the training data is diversified by assigning weights to training samples, but they usually leverage a single type base learner.

Homogeneous ensembles use the same feature selection method with different training data and distributing the dataset over several nodes.

Popular methods like bagging and boosting generate diversity by sampling from or assigning weights to training examples but generally utilise a single type of base classifier to build the ensemble.

Heterogeneous ensembles consist of members having different base learning algorithms such as SVM, ANN, Decision Trees etc., which can be combined to form the predictive model.

Heterogeneous ensembles use different feature selection methods with the same training data.

A popular heterogeneous ensemble method is stacking.

3 major kinds of ensemble learning algorithms that aims at combining weak learners:

Bagging – often considers homogeneous weak learners, learns them independently from each other in parallel and combines them following some kind of deterministic averaging process

Boosting – often considers homogeneous weak learners, learns them sequentially in a very adaptative way (a base model depends on the previous ones) and combines them following a deterministic strategy

Stacking – often considers heterogeneous weak learners, learns them in parallel and combines them by training a meta-model to output a prediction based on the different weak models predictions

Ensemble Learning Methods: Bagging

Bootstrap AGGRegatING, or bagging for short, is an ensemble learning method that seeks a diverse group of ensemble members by varying the training data, with the two key ingredients:

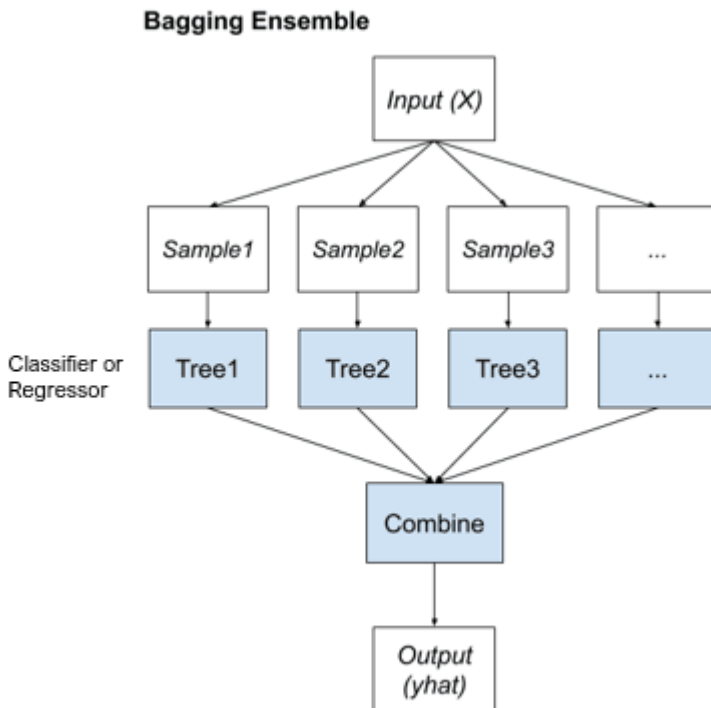
Bootstrap sampling often used in statistic with small datasets to estimate the statistical value of the data sample. Multiple different

training datasets can be prepared, used to estimate predictive model of the ensemble members, and make predictions.

Aggregation is to combine (ensemble parallelly) the predictions made by the ensemble members are then combined using simple statistics, such as voting or averaging.

Many popular ensemble algorithms are based on bagging approach, including:

- Random Forest
- Bagged Decision Trees (canonical bagging)
- Extra Trees



Ensemble Learning Methods: Boosting

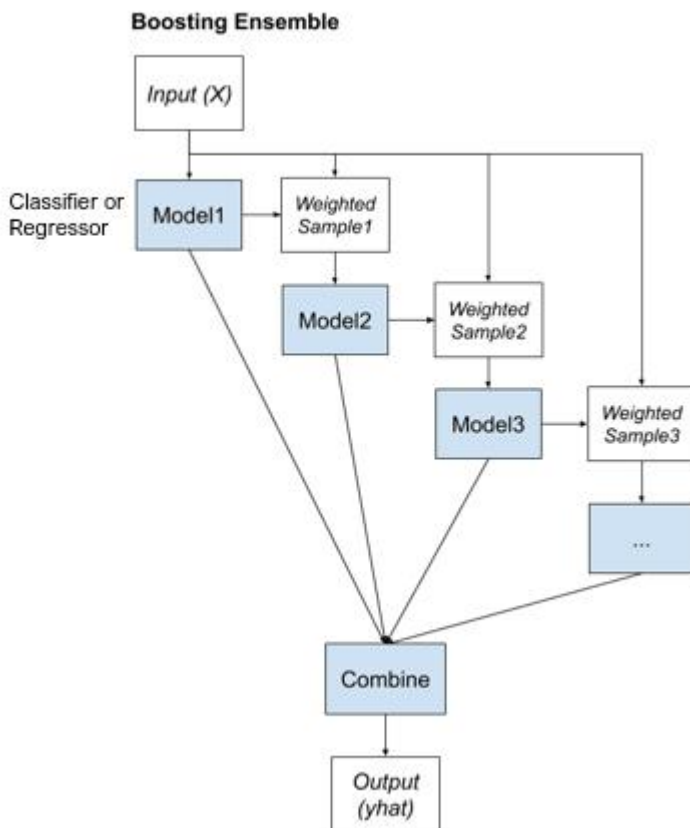
Boosting is an ensemble method that seeks to change the training data to focus attention on examples that previous fit models on the training dataset have gotten wrong (either misclassified or misestimated).

The key property is the idea of correcting prediction errors. The models are fit and added to the ensemble sequentially such that the second model attempts to correct the predictions of the first model, the third corrects the second model, and so on.

Learner predictions are then combined with voting mechanisms in case of classification or weighted sum for regression, generally the bagging mechanism.

Many popular ensemble algorithms based on boosting approach, including

- AdaBoost
- Gradient Boosting Machines (GBM)
- XGBoost (Stochastic Gradient Boosting)



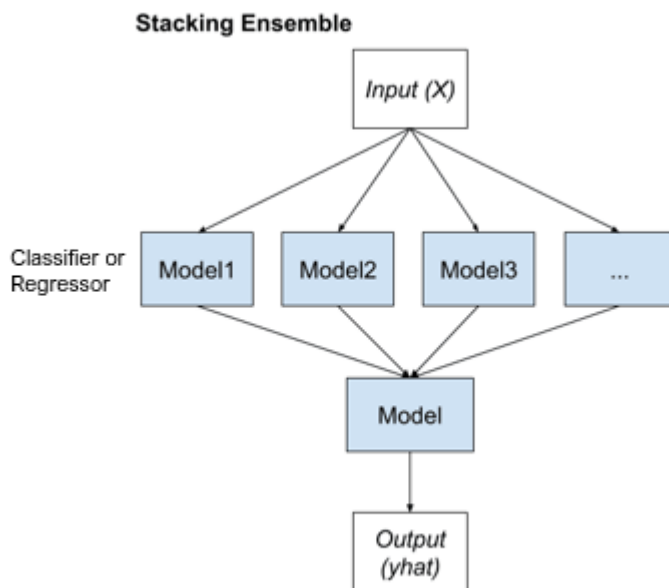
Ensemble Learning Methods: Stacking

Stacked Generalization, or Stacking for short, is an ensemble method that seeks a diverse group of members by varying the model types fit on the training data and using a model to combine predictions.

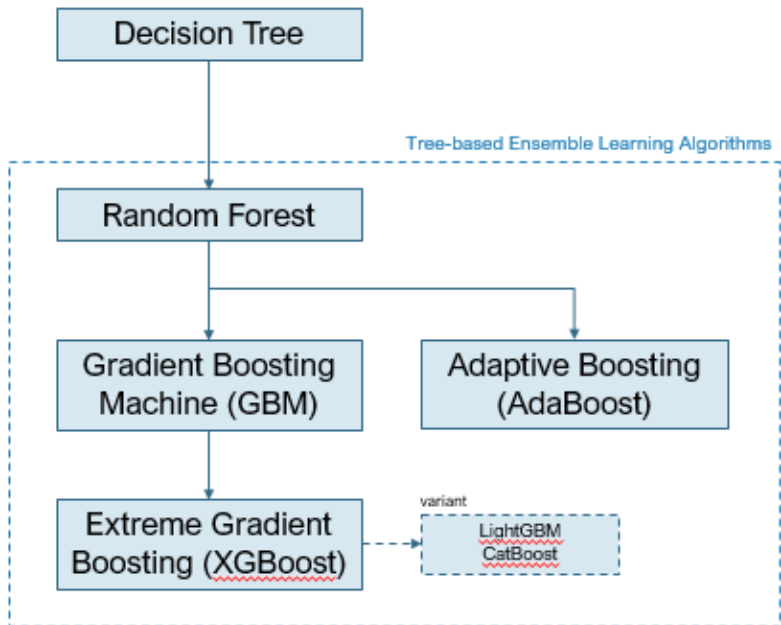
In Stacking, ensemble members are referred to as Level-1 models, and the model that is used to combine the predictions is referred to as a Level-2 model.

The 2-level hierarchy of models is the most common approach, although more layers of models can be used.

Multi-level hierarchy can be achieved by having 3 or 5 Level-1 models, and a single Level-2 model that combines the predictions of Level-1 models in order to make a prediction.



Tree-based Ensemble Learning Algorithms

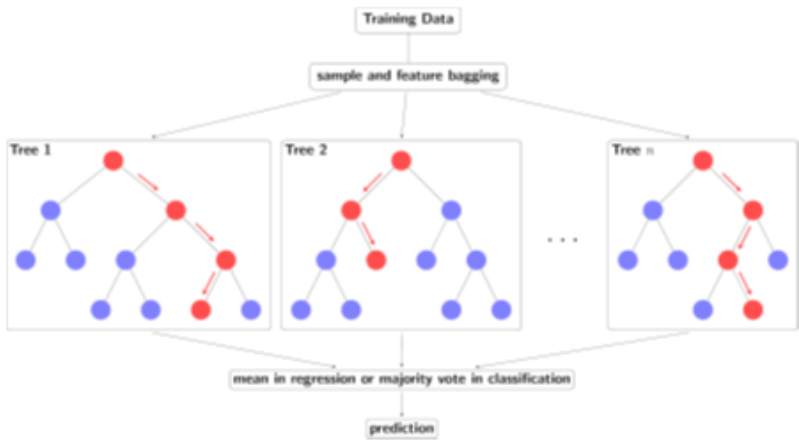


Random Forest

Random Forest algorithm is an ensemble of decision trees and a natural extension of bagging, which can be used for classification and regression.

Steps involved in random forest algorithm:

1. In Random forest, n number of random records are taken from the dataset having k number of records.
2. Individual decision trees are constructed for each sample.
3. Each decision tree will generate an output.
4. Final output is considered based on majority voting in classification or mean in regression.



Random forest is a collection of decision trees, still there are a lot of differences in their behaviour

Decision trees	Random Forest
Decision trees normally suffer from the problem of overfitting if it's allowed to grow without any control.	Random forests are created from subsets of data and the final output is based on average or majority ranking and hence the problem of overfitting is taken care of.
A single decision tree is faster in computation.	It is comparatively slower.
When a data set with features is taken as input by a decision tree it will formulate some set of rules to do prediction.	Random forest randomly selects observations, builds a decision tree and the average result is taken. It doesn't use any set of formulas.

AdaBoost (Adaptive Boosting)

AdaBoost (Adaptive boosting, also known as adaptive reweighting and combining algorithms) is a meta-algorithm formulated by Yoav Freund and Robert Schapire [1], originated from a research question of whether a set of weak classifiers could be converted to a strong classifier. Like any supervised learning machine, the goal of AdaBoost is to find a function that best

approximates the target variable y , from the values of the feature variables x :

AdaBoost is best used to boost the performance of decision trees where short trees of 1-level are constructed using split criterion like Gini as weak learners $h_t(x)$, called decision stump, and then combining the predictions as a weight sum of weak learners:

AdaBoost is designed specifically for binary classification with the following key logic:

A batch of training samples is created by bootstrap sampling.

The first-round classifiers (learners) are all trained using weighted coefficients that are equal.

In subsequent boosting rounds, the adaptive process increasingly weighs data points that were misclassified by the learners in previous rounds and decrease the weights for correctly classified ones.

The weak learners in AdaBoost are decision trees with a single split, called decision stumps (1-level decision tree).

AdaBoost works by putting more weight on difficult to classify instances and less on those already handled well.

Weight is updated for every data point.

The learning function is boosted by the weighted learners/classifiers.

Gradient Boosting Machine (GBM)

The AdaBoost framework was further developed by J.H. Friedman and called Gradient Boosting Machines (GBM) [1]. It is a robust ML algorithm made up of Gradient Descent and

Boosting. Just like AdaBoost, Gradient Boost also combines a no. of weak learners to form a strong learner.

Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting (XGBoost) was developed by Tianqi Chen and Carlos Guestrin [1] where the main different is that the GBM was further derived from gradient descent (a global optimisation) in function space into local optimisation using Newton's method [2] produced by repeatedly taking steps that are stationary points of the 2nd order Taylor series approximations [3] to the loss function (Hessian descent in function space).

The foundation of XGBoost algorithm is the 2nd order optimisation (shown here), and the other important element is the regularisation term, which was part of the original designed of the algorithm. It's efficiency and performance in learning non linear decision boundaries.

The XGBoost implementation (open-source package) supports both the L1 and L2 regularisation, and three other ways of controlling the model's complexity i.e. underfitting/overfitting, summed up as follows:

- Regularisation
- Pruning
- Sampling
- Early stopping

Classification ML algorithms: Definition

```
names = [
    "LinearSVM",
    "RBF SVM",
    "DecisionTree",
    "RandomForest",
    "AdaBoost",
    "GradientBoosting",
    "XGBoost",
    "LightGBM",
    "CatBoost",]

classifiers = [
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    AdaBoostClassifier(),
    GradientBoostingClassifier(),
    XGBClassifier(verbosity=0),
    LGBMClassifier(),
    CatBoostClassifier(verbose=0),]

plot_classification_comparisons(figure_size=(32, 9))
```

Example: Diabetes Patient Prediction

The Pima Indian are a group of Native Americans living in an area consisting of what is now central and southern Arizona.

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases.

The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset.

Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

The datasets consists of several medical predictor variables and one target variable (Outcome)

Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

Our objective is to develop a ML model for predicting diabetes patient – classification problem.

We will be evaluating almost all the ML classification algorithms we learned so far, plus introducing a new method called ensemble voting using the VotingClassifier.

We will also be using a new graph plotting tool called plotly that is beautiful and easy to use.

We will also be using natsort for sorting task.

```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

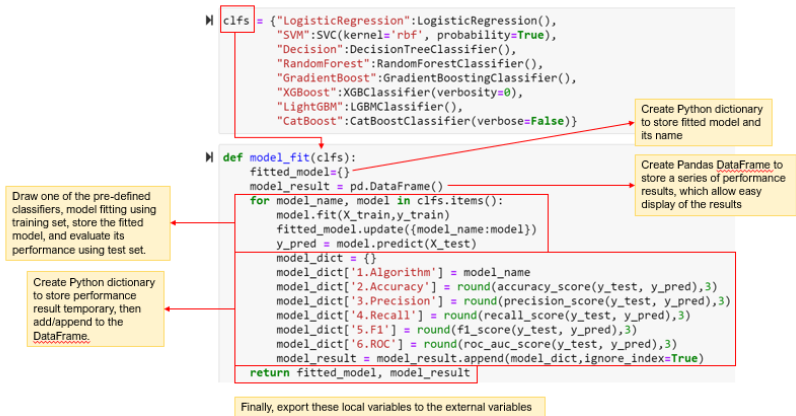
from sklearn.svm import SVC
from sklearn.gaussian_process.kernels import RBF
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import VotingClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier

from sklearn.utils import resample
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score
from sklearn.metrics import precision_score, recall_score, roc_auc_score

import missingno as msno
from natsort import index_natsorted

# Plotly graphic Library
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots

import warnings
warnings.filterwarnings('ignore')
```



Model Evaluation

